WildStar–Discovering an AmigaDOS 2.0 Hidden *SECRET*

AC's TECH / AMIGA

For The Commodore

Developing an AmigaDOS 2.0 Command Line Utility

Plus:

- CAD Application Design–Part II
- Programming the AMIGA's GUI in C–Part II
- Hash Tables for the Masses
- System Configuration Tips for SAS/C

...and much more!

ON DISK BONUS

**VBRMON
ASSEMBLY
LANGUAGE
MONITOR**

Amazing
Covers the AMIGA

Amazing Computing
For The Commodore AMIGA

AC's TECH
For The Commodore AMIGA

AC's GUIDE
To The Commodore AMIGA

AC's publications have always been innovative and complete. With the Premiere issue of ***Amazing Computing*** in February 1986, we introduced the first monthly magazine dedicated to the Amiga. AC's commitment to deliver solid information and valuable insight for the Amiga continues today. AC remains the first in news coverage—often providing complete stories and pictures of fast-breaking Amiga events in the next issue. AC is a forerunner in providing a well balanced mix of reviews, tutorials, tips, programming tasks, hardware projects, and more. Each issue of *Amazing Computing For The Commodore Amiga* is packed with the best of the Amiga.

AC's TECH For The Commodore AMIGA is the first and largest publication dedicated to the technical promise of the Amiga. Each quarterly issue provides new frontiers for the Amiga user eager to do more. *AC's TECH* not only attempts to define what the Amiga can do, but expands those boundaries.

AC's GUIDE To The Commodore AMIGA is the first and only complete guide to the Commodore Amiga. *AC's GUIDE* is the one resource used by the entire Amiga industry for Amiga product information. Yet *AC's GUIDE* also offers a listing of freely redistributable software and a growing registry of Amiga user's groups. *AC's GUIDE* is the complete resource to the expanding platform of Amiga products and services.

If you are not an AC subscriber, you don't know what you're missing. AC's publications are produced to give you more choices and resources. AC makes sure that whatever is happening in the Amiga market, you'll know about it.

To order a subscription, please use the order forms in this issue or for credit card orders, call toll-free
1-800-345-3360
from anywhere in the U.S. & Canada.

Contents   Volume 1, Number 3

Departments

```
printf("Hello");

print "Hello"

JSR printMsg

say "Hello"

writeln("Hello")
```

Whatever language you speak, AC's TECH provides a platform for both gaining insight and sharing information on its most innovative implementation for the Amiga. Why not see if your latest programming endeavor can help a fellow Amiga user expand upon his or her vocabulary? To be considered for publication in AC's TECH, submit your technically oriented article (both hard copy & disk) to:

AC's TECH Submissions
PiM Publications, Inc.
One Currant Place
Fall River, MA 02722

Startup-Sequence

Opportunity Knocking

AC's TECH is designed to educate, promote new technologies, and encourage the exchange of ideas in Amiga computing. As such, it is a rallying point for Amiga development. Professional Amiga developers and weekend programmers alike have experienced the thrill of a completed project. We know what it means to create a program or build a piece of hardware. On this basis, I would like our readers to consider focusing their talents on creating Amiga products.

If you have ever had the desire to program a new product or create a piece of hardware for the Amiga, now is the time. There has never been such an opportunity to excel in the marketplace. Not only is Commodore more committed than ever to seeing that the Amiga is accepted as a professional machine, but there is now an entirely new market in CDTV.

Yes, Commodore Is Marketing

Commodore's efforts in marketing have never received great praise from Amiga users. Many of us believe that Commodore could have done more to position the Amiga in the U.S. market. However, CBM has made a new commitment to carry the Amiga and CDTV into mainstream use.

CommodoreExpress is CBM's year-old program for Amiga 500 users. If your A500 fails and a call to CBM does not provide a remedy to the problem, Federal Express will pick up the machine at your home or office. It will be repaired and returned within three days (just remember to keep your original packing material). This program will also be available to CDTV users.

Also available under CommodoreExpress is the Gold Card Service program for Amiga 2000 and Amiga 3000 owners. Commodore will provide a one year warranty with on-site service for these professional Amiga systems. There is also an option to purchase an additional one to two years of on-site service. The additional confidence these programs can provide for anyone on a tight schedule is priceless.

In addition, CBM has initiated a leasing program for the Amiga. Through Commodore's lease arrangement, most businesses will find it easy to incorporate Amigas into their facilities.

In order to use the service and customer satisfaction programs, Commodore must attract the customer. Commodore will concentrate on print media with specially focused Amiga and CDTV advertisements to run in six national magazines, such as *People* and *Newsweek*, this fall. There will also be spot television advertisements.

Included in a brochure supplied by Commodore are a few examples of the proposed advertisements. Commodore appears to be aggressively pushing the Amiga into the home education market, with particular emphasis being placed on the use of computers by children.

What does it all mean? Commodore is going to do what we have always asked them to do. They are going to sell Amigas. And, although there is a wide assortment of software and hardware currently available for the machine, there is more to be done.

Enter CDTV

Not excited yet? Think about CDTV. CDTV is a brand new market based on the Amiga technology. Almost all the tools available to program and create on the Amiga will work for CDTV. It also offers software developers a world of options from sound to graphics.

We always knew we could tap the CD in CDTV for great sound quality. Now Commodore has released CDXL, which allows a software developer to pull full-motion video from the CD into a screen one-third the size of the normal display in real time. This is now accessible to CDTV developers for use in games, educational software, or any area where full-motion video can be an asset.

As CDTV matures and sales increase, there will be a large block of consumers hungry for good software and hardware. The people who have a responsibility in securing this market are the Amiga product designers. With 540 megabytes of storage and an entire world of ideas to explore, CDTV offers an opportunity that has never been seen before. Add to this Commodore's commitment to make CDTV a standard and the possibilities are endless.

Yes, *AC's TECH* is supposed to excite you into the possibilities of creating on the Amiga. But with all of this currently available, our job is extremely easy.

Sincerely,

Ernest P. Viveiros, Jr.

SOME WORDS OF PRAISE FOR BASIC AND THE AMIGA

Dear AC's TECH:

The April 1991 issue of BYTE magazine has given some noteworthy recognition to both the Amiga and BASIC as an environment for learning and experimenting in computer science. The article "A Fast, Easy Sort", by Stephen Lacey and Richard Box, presents a sorting technique called "Combsort", an alternative to the much known, intuitive, but incredibly slow, bubble sort. The authors did their work on an Amiga 2000! Their sorting algorithm is very well described and I recommend the article to programming enthusiasts of all levels. Example code is given as a single subroutine in both C and True BASIC. To experiment with it I had to design my own driver program.

True BASIC is an integrated environment from Kemeny and Kurtz, the original inventors of BASIC at Dartmouth College in New Hampshire. It is a fully structured high level language (like C and Pascal) that is both powerful and easy to use. True BASIC programs are completely transportable (including graphics!) to four different microcomputers, as well as a number of others in the mini and mainframe class. I routinely develop programs on my IBM and execute them on my Amiga, and visa versa.

On the magazine diskette (this issue) you will find programs that demonstrate the above mentioned Combsort subroutine, as well as its Bubble Sort equivalent, in three different languages: AmigaBASIC, True BASIC, and finally SAS/C. (Note that both AmigaBASIC and True BASIC do not yet execute properly on a 32 bit machine, although I hear that True BASIC will be upgrading to do that very soon) My programs take the BYTE magazine article one step further and generate a graphic representation of the program's execution. Initially, a low resolution screen is filled with dots representing 320 randomly chosen numbers (312 in AmigaBASIC, crazy eh?). Then the program arranges them in ascending order, resulting in a rising graph of sorted values. It is instructional to watch the computer slowly sort out the numbers before your eyes. For the C and True BASIC versions I have included the corresponding stand alone (compiled and linked) programs, allowing you to execute them even if you do not own either language.

These programs provide an excellent example of the comparative advantages of the three development environments for doing investigative work in computer science on the Amiga. Although the C version is faster than both versions of BASIC, it requires that the programmer be familiar with many system dependant details, and is perhaps beyond the level of the average computer hobbyist. Check out the code yourself.

The AmigaBASIC version shows a big reduction in complexity, but it has structural limitations due to the fact that external subroutines cannot be called by other external subroutines. This limits how well you can structure programs, affecting their readibility. Also, AmigaBASIC does not support screen scaling. Vertical pixel numbers are upside down, increasing from top to bottom, rather than the normal bottom to top of Cartesian Coordinates (or world coordinates, as we say in computer science). I compensate for this by including a user defined, vertical screen scaling function. Horizontal scaling is not required in this particular example, but to be fair I should point out that most graphic programs would require it.

All three programs were designed using the most similar hierarchical structure possible in each language. External subroutines (separate fragments having their own local variables) were used in both versions of BASIC in order that they more closely resemble their C counterpart, which treats all functions as external. True BASIC demonstrates itself as the most readable (pseudocode-like), shows no structural limitations, and is reasonably fast.

LANGUAGE	BUBBLE SORT	COMBSORT
AmigaBASIC	21 min, 34 sec	1 min, 10 sec
True BASIC	11 min, 22 sec	0 min, 30 sec
C	0 min, 45 sec	0 min, 2 sec

Note that these programs do not represent a 100% accurate comparison because of the time required for the system to write pixels every time a swap occurs. If you want to make your own accurate measurements of this sorting method you should use the straight sorting code from the original article. Still, these graphic versions are accurate enough for instructional purposes, and they are fun to look at.

I was happy to see such an article. A certain number of us have purchased our Amigas in order to learn and experiment with programming, and BASIC does indeed play a large role in that. I encourage similar articles here in AC's TECH/AMIGA.
—Paul Castonguay

CAD
application design

Press the right mouse button—a menu pops up. Select an action command from the menu, move the mouse cursor over an object and press the left mouse button—the object changes color. Move the mouse again—the object moves, changes size, or rotates as the mouse moves. Press the left mouse button a second time—the highlighted object is erased and redrawn at its new location, size, or orientation. What was just described is a "point, click, and drag" metaphor used in modern CAD systems for translating, scaling, and rotating geometric objects. This segment of the series shows how to implement the point, click, and drag" metaphor when designing Amiga CAD applications

Introduction

How does a CAD program tell which object a user picked when a mouse button is pressed? How are mouse movements in pixel coordinates converted into world coordinate translations, scale values, or rotation angles? In this article, we'll see how CAD programs do all these things and more, and develop procedures to implement the metaphor just described. We'll also develop procedures for "panning" a drawing which is too large to fit in a window, and "zoom in" to reveal small drawing details or "zoom out" to show a bigger view of the drawing. The procedures will then be put to work in an event-driven program which will let us move, resize, and rotate objects, or pan and zoom our model world using just the mouse.

Actions, Objects, and Modifiers

When we are interacting with a program, we are telling the program what is to be done and which objects are to act or be acted upon. We are giving the program, or objects in the program, commands. Since natural language sentences are constructed from nouns and verbs, an effective way to allow a user to input program commands is with sentence fragments containing nouns and verbs. The verbs describe program actions, and the nouns describe the objects which are to be manipulated. The sentence fragments can take one of two forms: either "action-object" sentences, or "object-action" sentences. Commands can also contain object modifiers, action modifiers, or both. As an example, a typical command for deleting

all lines in a drawing program might be either "Delete-all-lines," or "line-delete-all." The action is "delete," the objects are "lines," and the modifier is "all." Many commands don't require that an object be specified: the object is assumed to be whatever object has already been selected, or assumed to be the next object which will be selected. These types of commands require two user actions. The user selects an object and then an action (pre-select mode), or selects an action and then an object (post-select mode). Many commands don't require any object specifiers at all. "Quit," "Save," and "Delete-All" are examples. These are "immediate mode" commands, and require only a single user action.

Intuition and other graphical interfaces provide tools which can be used to elicit user commands. The primary tools are menus, gadgets, requesters, and pointing devices. When we write interactive programs, we have to decide what commands our programs will recognize, then match the commands to the graphical interface tools which are available to us. Sometimes, an entire command can be mapped to a single tool (a window "close" gadget, for example), but many times, we have to break a command into parts and match the parts to different interface tools. When we break a command into parts, our program has to put the parts back together before it can make sense of the command.

PART II—INTUITION EVENTS AND USER INTERACTION

BY FOREST W. ARNOLD

Interactive tools such as menus and gadgets let us write programs which are simple to use (if the interface is well designed). However, these same tools also increase the complexity of a program's code. When a user picks a menu item, Intuition doesn't send our programs whatever command the menu item represents. Instead, Intuition sends us a MENUPICK event, and our program has to figure out which menu item was picked, and then figure out what command the menu item represents. To manage this task, we need general-purpose techniques for receiving and processing commands through input events which will make our program code as simple as possible.

Event Driven Programming

If an Intuition window is opened with one or more IDCMP flags set, Intuition automatically creates an "Intuition Direct Communication Message Port" for the window, and as input events occur, places event messages in a queue attached to the message port. Input events can be generated by a program user with the mouse or the keyboard. They can also be generated by other applications, or by devices such as the trackdisk device or timer device. Programs which receive input through the IDCMP "wait" until notified that an event message has arrived, read the message, perform whatever action is appropriate, and reply to the message. Programs which wait for an event to occur before "doing something," and which only do something when an event occurs, are called event-driven programs.

The process of just responding to input events seems simple. Many times it is, but most of the time it's not. There are two reasons why event-driven programs

can be complicated. First, interactive programs can't usually determine or predict the order in which input events will arrive. Users, especially inexperienced ones, don't always do things in the order programmers expect them to. Second, most non-trivial program actions require a sequence of events to occur before a program can determine what needs to be done and to gather the information needed to do it. As an example, before an object can be moved in our demonstration program, several related events have to occur in sequence. The "move" action needs to be specified with a menu pick, the object to be moved must be picked with the mouse select button, and the move values need to be obtained by tracking the position of the mouse until a second select button press occurs. An object can not be moved until all of these events have occurred. The program code required to take care of unexpected events and to keep up with a series of related events can quickly become complex and unmanageable.

Fortunately, the complexity of event-driven programs can be reduced through the use of "event handlers" and "state vectors." State vectors are data structures used to keep up with the current state of a program and its data. Event handlers are procedures which are called when a specific event or class of events

occurs. Event handlers are simple procedures which only do one of two things: they either directly perform an action, or they modify a program's state. Actions, in turn, usually modify an event-driven program's state, and state changes may also trigger actions. Most event handlers which modify a program's state generally evaluate the new state to determine if an action should be triggered.

When event handlers and state vectors are used, program logic is moved out of a program's input code and distributed among its event handler procedures. The top-level input logic for event-driven programs is simple: the input code just calls an event handler and sends it whatever data it needs. The algorithm we'll use for receiving and dispatching input events is:

- Receive an event
- Retrieve the event handler which will process the event
- Retrieve the data needed by the event handler
- Call the event handler, sending it the data it needs

→

Definition of the Extension Structure

```
typedef int (*eventProc_t)(struct IntuiMessage *,void *);

typedef struct _intuiExtension
{
  eventProc_t   handler;    /* pointer to event handler  */
  void          *data;      /* arg for event handler     */
} intuiExtension_t;
```

To implement this algorithm, we need to associate event handlers and their data with input events. The simplest way to do this is with a structure containing pointers. We'll define an "Intuition extension" structure to hold both a function pointer to an event handler and a pointer to the handler's data. Our extension structure will then be attached to whatever Intuition object which generates events that the handler is responsible for processing. The definition of the extension structure we'll use (we also define a type for event handler function pointers to make casting the pointers easier) is shown on page 9.

When an event arrives, the input handler retrieves the extension structure, gets the pointer to the event handler and the pointer to the handler's data, and calls the event handler using the function pointer. This process continues until an event handler returns a "failure" code, or a "stop the program" code. But how do we attach the extension structure to an Intuition object and retrieve it in the input handler?

Almost all Intuition input events occur in windows, gadgets, menus, or menu items. Intuition's Window structure and Gadget structure contain generic pointer members named "UserData." For windows and gadgets, we can use the UserData member to hold a pointer to our extension structure. Unfortunately, Menu and MenuItem structures don't have "UserData" pointers. To attach our extension pointer to menus or menu items without using some kind of look-up table, we have to "overload" Intuition's structures. We do this by defining our own menu item structure which has Intuition's structure as its first member. Here is the typedef for a structure we can use to overload Intuition's MenuItem structure:

```
typedef struct _myMenuItem
{
  struct MenuItem mi;   /* Intuition's MenuItem   */
  intuiExtens_t  extens; /* our extension          */
} myMenuItem_t;
```

Since Intuition's MenuItem structure is the first member of myMenuItem_t structures, myMenuItem_t structures can be safely cast to MenuItem structures. If (and only if!) all of our menu items are created as myMenuItem_t structures, MenuItem structures can also be safely cast to myMenuItem_t structures.

To retrieve our extension structure pointer inside an input handler, we examine the event message to determine what input object the message applies to. If it applies to a menu item, we get the address of the menu item, cast it to "myMenuItem_t*", and then get the pointer to our extension. If it applies to a gadget, we get the pointer to the gadget by casting the "IAddress" member of the IntuiMessage structure to "Gadget *", then get the pointer to our extension by casting the gadget UserData member to "IntuiExtension_t*". The extension pointer is retrieved from a window pointer in the same way. You can look at the procedure handleInput( ) in the demonstration program to see how these pointers are cast, retrieved, and used. You can also see how simple the code for the input handler is.

Commands and Events

Event-driven programs receive commands through events, so the commands a program recognizes have to be mapped to events and to interface objects which generate events. Simple commands can be mapped to a single interface object, and received as a single event. Complex commands

Table One
Demonstration CAD Program Commands

Command	Syntax	Mode
Quit project	Action	Immediate
Select an object	Action - object	Immediate
Drag an object	Action - object	Immediate
Move an object	Action - object	Post-select
Size an object	Action - object	Post-select
Rotate an object	Action - object	Post-select
View - All	Object - action	Immediate
View - Zoom - In	Object - action	Immediate
View - Zoom - Out	Object - action	Immediate
View - Zoom - Box	Object - action	Post-select
Pan (World)	Action - Object	Immediate
Close Window	Action - object	Immediate
Resize Window	Action - object	Immediate
Redraw Window	Action - object	Immediate
Move Window	Action - object	Immediate

Table Two
BREAKDOWN of Demonstration CAD Program Commands

Command	Interface Object	Event	Event Handler
Quit	MenuItem	MENUPICK	miQuit( )
Select	Select button	MOUSEBUTTONS	windowEvent( )
Drag	Mouse	MOUSEMOVE	multiple
Move	MenuItem	MENUPICK	miSetAction( )
Size	MenuItem	MENUPICK	miSetAction( )
Rotate	MenuItem	MENUPICK	miSetAction( )
View all	MenuItem	MENUPICK	miFullView( )
Zoom in	MenuItem	MENUPICK	miZoom( )
Zoom out	MenuItem	MENUPICK	miZoom( )
Zoom Box	MenuItem	MENUPICK	miSetAction( )
Pan	Prop Gadget	GADGETUP	panView( )
Close	System Gadget	CLOSEWINDOW	windowEvent( )
Resize	System Gadget	NEWSIZE	windowEvent( )
Redraw	Intuition	REFRESHWINDOW	windowEvent( )
Move Window	System Gadget	automatic	

usually need to be broken up, the pieces mapped to two or more interface objects, and then received as multiple events. Inside the program, the events have to be interpreted as commands, or pieced together into a complete command. Event handlers are the program procedures which interpret and implement commands by performing an action, or by piecing commands together from information in a program's state vector.

Let's look at the commands in our demonstration program to see how they are implemented through events. The commands, their syntax, and their modes are shown in Table One

Which Intuition interface objects do we map our commands to? The "Close," "Resize," and "Drag" system gadgets are clear choices for the "Close Window," "Resize Window," and "Move Window" commands. The mouse select button is used to select objects, and the mouse cursor is used for dragging objects. Gadgets, menu items, and menu subitems are good interface objects to use for specifying objects, actions, or modifiers. I prefer using gadgets for simple, immediate commands, and menu items and subitems for complex commands. The demonstration program's commands and their corresponding interface objects, events, and event handlers are shown in Table Two.

Let's take a look at what each handler in the demonstration program does, and where each gets the data it needs:

WindowEvent( ) processes all MOUSEBUTTONS, NEWSIZE, REFRESHWINDOW, and CLOSEWINDOW events. For MOUSEBUTTONS events, windowEvent( ) converts window coordinates to world coordinates and calculates the world coordinate distance corresponding to three pixels in x or y, whichever is largest. These values are saved in the state vector. The state vector is examined to see if an object has already been selected. If not, windowEvent( ) calls findObject( ) to search for an object located within three pixels of the mouse "pick" point. If an object is found, its pointer is placed in the state vector. Finally, if an "action" procedure pointer is in the state vector, windowEvent( ) calls the procedure. When CLOSEWINDOW events are received, windowEvent( ) just returns a non-zero value to handleInput( ), which causes it to stop processing input and return to main( ). This is one of the ways the demonstration program is stopped. For REFRESHWINDOW events, windowEvent( ) calls drawAll( ) to redraw the model world. If a NEWSIZE event is received, windowEvent( ) calls setPanGadgets( ) to adjust the size of the pan gadgets. Note that resizing a "smart refresh" window will generate a NEWSIZE event, but won't generate a REFRESHWINDOW event unless the window is made larger. WindowEvent( ) gets all of the data it needs from the program's state vector, the window pointer, and the IntuiMessage event structure.

The demonstration program has only two gadgets. One is a horizontal "scrollbar" used to pan the world view left or right. The other is a vertical scrollbar used to pan the view up or down. The event handler for both gadgets is panView( ), which is called when a GADGETUP event is generated. When panView( ) is called, it updates the window's view transform and redisplays the model world. The data needed in panView( ) is obtained from the program's state vector, the IntuiMessage event structure, and the gadget pointer.

MENUPICK events are handled by several different event handlers. Which one is called depends on which menu item is selected. As mentioned above, menus and menu items are good interface objects for setting up complex commands. Combinations of actions, objects, and modifiers can be attached to menu items, and then sent to event handlers when the menu items are selected. To send all of this information to a menu item event handler, we define an "miData" structure to store all the data in one place. The typedef's for the "action" procedure and the "miData" structure are shown above.

A pointer to the miData_t structure for a menu item is placed in the intuiExtension_t structure attached to the menu item. This becomes the "data" pointer which is sent to a menu item event handler when it is called from the event input procedure. The menu item event handlers in the demonstration program are miQuit( ), miFullView( ), miZoom( ), and miSetAction( ).

MiQuit( ) is called when the "quit" menu item is picked. MiQuit( ) returns a non-zero value to handleInput( ), causing it to stop processing input and end the program. MiQuit( ) doesn't require any data at all to do its job.

The "View-all" menu item command means "show me a full view of the world". This is an immediate mode command and it is implemented in miFullView( ). MiFullView( ) calls fullDisplay( ) to find the extent of the world coordinates, calculate a new view transform which will fit the entire model world centered in the Intuition window, and redisplay the world data. The only data miFullView( ) needs is a pointer to the state vector and a pointer to the Intuition window.

The menu item handler for both the "Zoom-in" and "Zoom-out" menu items is miZoom( ). "Zoom-in" and "Zoom-out" are good examples of commands with action modifiers. The action part of the commands is "Zoom" and the modifiers are "in" and "out." Zooming is accomplished by rescaling the window's view transform, then redisplaying the model world. In the demonstration program, when the zoom-in menu item is picked, the view scale factor is multiplied by two. This causes the model world to be redisplayed at twice its previous size. Likewise, to

→

zoom out, the view scale factor is multiplied by one-half, which causes the model world to be redisplayed at half its previous size. Since miZoom( ) is called for both the "zoom-in" and "zoom-out" menu items, a ZOOMIN or ZOOMOUT modifier is placed in the miData_t modifier field so that miZoom( ) can determine which way to zoom the view. The only other data miZoom( ) needs is an Intuition window pointer, which it gets from the IntuiMessage structure.

MiSetAction( ) is called when the "move," "size," "rotate," or "zoom-box" menu items are picked. MiSetAction( ) places a function pointer to the action procedure which will process the action part of the command in the program's state vector. It also places modifiers and pointers to the data the action procedure will need into the state vector. Basically, miSetAction( ) is placing part of a command into the state vector for later use. When an object is selected (in windowEvent( )), the command will be complete and the action procedure will be called. The data which miSetAction( ) needs is an action procedure function pointer, an action or object modifier, and any other data the action procedure will require to do its work. This information is obtained from the miData_t structure pointer sent to the menu item event handler. The action procedures sent to miSetAction( ) are dragAndMove( ), dragAndSize( ), dragAndRotate( ), and dragZoomBox( ). The corresponding MiData_t structures are moveData, sizeData, rotateData, and zoomBoxData. You can look at the program code to see how these structures are defined, initialized, placed in the myMenu_t structure for each menu item, and then placed in the state vector by miSetAction( ).

As you can see, attaching event handlers and data to Intuition input objects simplifies event processing and makes the logic in our event handlers pretty simple! We've seen how the the event handlers communicate information to each other using the program's state vector and decide what to do from information in the state vector. Now let's see what the state vector looks like, and see which procedures set and use its members. In the demonstration program, the state vector is called "world," and it is a global instance of the following structure:

```
typedef struct _world
{
    double minx,miny;      /* world extent lower left    */
    double maxx,maxy;      /* world extent upper right   */
    actionProc_t action;   /* action procedure pointer   */
    object_t *selected;    /* selected object            */
    void    *data;         /* action procedure data      */
    long    modifier;      /* action-object modifier     */
    long pickX,pickY;      /* select pick coordinates    */
    double wrldX,wrldY;    /* world coordinate pick pt   */
    double eps;            /* world search distance      */
    object_t *objects;     /* list of world objects      */
    transform2_t *viewTf;  /* current view transform     */
    double  aspect;        /* window aspect ratio        */
} world_t;
```

The "action," "data," and "modifier" members are set in miSetAction( ) when an "Action" menu item or the "Zoom-box" menu item is picked. The action member is used in windowEvent( ) to determine if the action part of a command has been specified, and if so, to call the action procedure. The data and modifier members are general-purpose members used either by menu item event handlers or by "action" procedures.

The "selected," "pickX," "pickY," "wrldX," "wrldY," and "eps."members are set in windowEvent( ) and are used in findObject( ) and the "drag" action procedures. "Selected" points to whatever model object is within a distance of "eps" of the mouse pick coordinates, "pickX" and "pickY." The pick distance, "eps," is calculated each time windowEvent( ) is called. Since the world distance corresponding to our pixel pick distance (3 pixels) will change when the view transform changes, "eps" needs to be recalculated each time we search for an object. "WrldX" and "wrldY" are the world coordinates which correspond to the mouse pick coordinates. These are calculated by transforming the pick pixel coordinates.

The "objects" and "viewTf" members are used by any procedure which needs to access the linked list of world objects or to transform coordinates. The world coordinate extent members ("minx," "miny," "maxx," "maxy") and the view transform member ("viewTf") are set in fullDisplay( ) when it is called by miFullView( ). The extent members are also updated whenever objects are moved, resized, or rotated. The extent of the world is always the smallest rectangle which completely contains all the world objects. As objects in the model world are moved, rotated, or resized, the size of the world changes. This method of dynamically sizing our "drawing area" is different from the method used in many CAD systems. Usually, the size of the drawing area for a model world must be set up in advance, and can not be easily changed later on. Dynamically sizing the world removes artificial program limits on the size of a drawing.

Although the demonstration program accepts only a few commands, the techniques it uses to interpret events as commands and process the events can be used in any event-driven program, no matter how complex. Event-driven programs are like "Rube Goldberg" machines: a simple little event sets off a whole chain of actions that state changes, which eventually result in the program doing what the user wanted it to do. The trick in event-driven programming is to build the chain by linking simple parts together with event handlers and a state vector. After you develop a "feel" for visualizing how simple actions can be chained together using the above techniques, you'll be able to easily write event-driven programs which can handle almost any command.

Here are the main points to remember about the technique:

1. *Program commands are placed in "action-object" or "object-action" sentence fragments.*
2. *Command actions, objects, and modifiers are mapped to graphical interface tools which generate events. Complex commands are mapped to multiple interface tools.*
3. *Event handler procedures which implement command actions or modify a program's state are attached to graphical interface tools, and called using function pointers when the tools are picked.*

4. *The state information and data for event-driven programs is kept in a state vector. Event handlers communicate data and control information to each other using the state vector, and use information in the state vector to piece together complex commands.*

Drag Interaction

Up to this point, we have seen how the top-level control flow for our program is implemented with event-driven programming techniques. The real work in our program is done by the procedures which enable us to move, scale, or rotate world objects with the mouse, and to pan and zoom the view window. The action procedures used to transform world objects by dragging them with the mouse are dragAndMove( ), dragAndSize( ), and dragAndRotate( ). These are the procedures whose pointers are placed in the state vector by miSetAction( ), and called by windowEvent( ) when an object is picked with the mouse select button. The action procedure for the "Zoom-box" menu item is dragAndZoom( ). All four of these procedures use "drag" interaction (also called "rubber-banding") to obtain transform values.

The basic drag technique is to continuously erase and redraw a copy of an object as the cursor moves. When the drag "end" event occurs, the overall change in the cursor's position is used to calculate the change in the object's position, size, or orientation. The object is then erased from its original position, its new position is calculated, and it is redrawn. To avoid erasing anything which is under the object as it is being dragged, all drag drawing is done in COMPLEMENT (exclusive-or) mode. Complement mode inverts bits in a screen or window's bitplanes: any bit that is "1" becomes "0", and vice versa. If a line is drawn in complement mode, then drawn again at exactly the same location, the original bit pattern is restored. Effectively, the object is erased without erasing anything under it.

Since all drag procedures work the same way, we can write a single, general-purpose "drag handler" procedure and use it for any drag interaction. The drag handler in the demonstration program is handleDrag( ). It processes MOUSEMOVE input events and uses a function pointer to call a drawing procedure to erase, move, and redraw the object being dragged. It is called from the four "action" procedures. The algorithm for handleDrag( ) is shown in table three.

The algorithm is pretty simple, but there are a few details we need to take care of to make it work correctly. First, if a MENUVERIFY event is received (menu button pressed) while an object is being dragged, we need to set the input message "Code" member to MENUCANCEL before replying to the message. This lets Intuition know we don't want a menu to be popped up while an object is being dragged. Second, after the input loop ends, we need to make sure the object has been erased and redrawn at its final position before we call dragDraw to end the drag. Otherwise, since the object is being erased by being drawn in COMPLEMENT mode, the draw and erase will get out of sync and the object won't actually be erased. Finally, for the procedure to work, we have to let Intuition know the window is to receive MOUSEMOVE events. We do this by setting the FOLLOWMOUSE flag in the NewWindow structure's "Flags" member when the window is created.

Let's look at the demonstration program's "dragDraw" procedures to see how they interact with handleDrag( ). The procedures are moveDrag, sizeDrag( ), rotateDrag( ), and zoomDrag( ). They all perform the same steps:

Erase the drag object by drawing it in COMPLEMENT mode.
If the drag code is BEGINDRAG or ENDDRAG,
    return.
Else
    Use move delta values to calculate object's new position, size,
    or rotation.
    Draw the drag object in COMPLEMENT mode.

Table Three

Algorithm for handleDrag( )

INPUT:	win - pointer to IDCMP window,
	object - pointer to object being dragged,
	startX,startY - starting mouse position,
	moveEps - amount by which mouse must move before dragDraw procedure is called.
	dragDraw - pointer to dragDraw procedure.
OUTPUT:	dx,dy - Overall change in mouse position.

Save window's foreground pen color, drawing mode, and IDCMP flags.
Set foreground color to window background color.
Set drawing mode to COMPLEMENT.
Modify IDCMP so window receives only MOUSEMOVE, MOUSEBUTTONS, and MENUVERIFY events.
Call dragDraw procedure to draw object at its initial position.
Do until SELECTDOWN event is received.
    Calculate overall mouse move (dx,dy) values.
    Calculate current mouse move values.
    If current mouse move values in x or y are greater than or equal to move epsilon value,
        Call dragDraw procedure to erase object.
        Move object, and redraw object.
End do.
Call dragDraw procedure to erase object from its final position.
Restore window's initial pen color, drawing mode, and IDCMP flags.

Object Containing Rectangle
(RX1, RY1)

Drag StartX, StartY
(RX2, RY2)

(X1, Y1)

(X2, Y2)

DY

DX

(RX3, RY3)

(RX4, RY4)

(X4, Y4)

Drag Rectangle

(X3, Y3)

```
DX = Mouse X - StartX
DY = Mouse Y - StartY
X1 = RX1 + DX,  Y1 = RY1 + DY
X2 = RX1 + DX,  Y2 = RY2 + DY
X3 = RX3 + DX,  Y3 = RY3 + DY
X4 = RX4 + DX,  Y4 = RY4 + DY
```
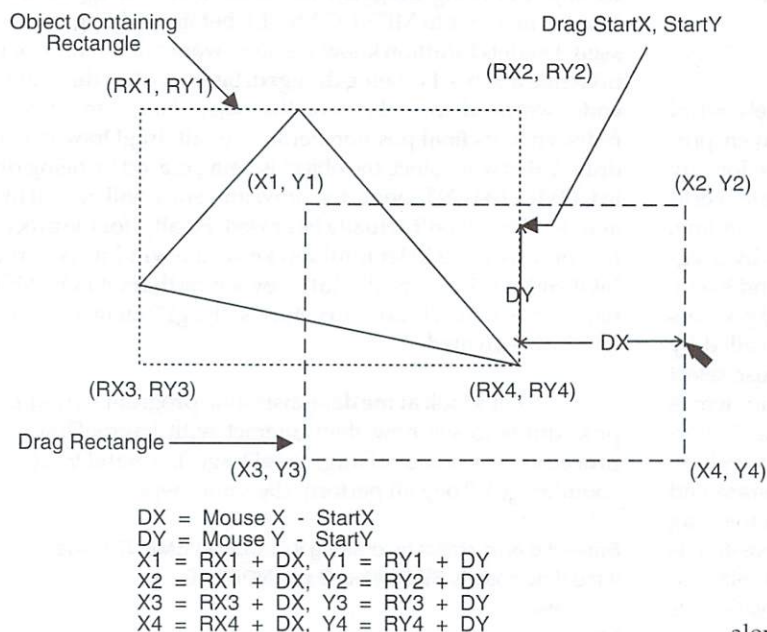
The object being dragged is not the actual world object which was selected. Instead, it is the smallest rectangle which contains the selected object. The rectangle data is set up by the drag action procedure which calls handleDrag( ). In the case of zoomDrag( ), there is no selected object. Instead, a rectangle is being created which defines a new view window.

MoveDrag( ) moves the rectangle by adding the change in coordinate values to its previous coordinate values. SizeDrag( ) interprets the delta coordinate values as changes in the width and height of the drag rectangle. The delta values are subtracted from, and added to, the rectangle's corner coordinates to change its size. ZoomDrag( ) is very similar to sizeDrag( ), except the rectangle is resized relative to its starting corner instead of its center. Figure One shows how the delta coordinate values are used to drag a rectangle, and Figure Two shows how the values are used to resize a rectangle.

The code for rotateDrag( ) is somewhat complicated because it has to calculate angles and use a rotation matrix to rotate the rectangle as it is being dragged. RotateDrag( ) uses the delta coordinate values to calculate the change in the rotation angle for the rectangle. The mouse cursor x and y coordinates are interpreted as the horizontal and vertical distances from the center of the rectangle. The rotation angle corresponding to these distances is found by calling the library function atan2( ). Figure 3 illustrates how a rotation angle is found using drag delta coordinate values. The previous rotation angle is subtracted from the new rotation angle to find the change in the rotation angle. This value is then used to incrementally rotate the rectangle around its center. To simplify the rotation code, the rectangle is created as a polygon using world coordinates. Incidentally, the Lattice documentation for the angle returned by atan2( ) is incorrect. The procedure returns an angle in the range -PI < angle <= PI, not -PI/2 < angle <= PI/2 as the documentation states.

Now that we know how handleDrag( ) and the "dragDraw" procedures work, let's see how the drag "action" procedures set up the data for the call to handleDrag( ), and then use the drag return values. Again, they all perform basically the same operations. First, the selected object is highlighted by drawing it in a "highlight" color. Next, the coordinates of the minimum containing rectangle (MCR) around the object are placed into an array. The array of rectangle coordinates is then sent to handleDrag( ), along with the rest of the data handleDrag( ) needs. After handleDrag( ) returns, the highlighted object is erased and the overall change in the position of the mouse is used to find the "move" translation value, the "size" scale value, or the "rotate" rotation angle. The object is transformed, its minimum containing rectangle values are updated, and it is redrawn by redrawing the entire window (this can be optimized so that only the object is drawn). DragZoomBox( ) is slightly different, since it is used to create a rectangle whose position and dimensions define a new view window for the view transform.

Object Containing Rectangle

(X1, Y1)

(X2, Y2)

Delta Y

Delta X

(X3, Y3)

(X4, Y4)

Drag Rectangle

Containing Rectangle Center (CX, CY)

```
DeltaX = ABS(Mouse X - CX)
DeltaY = ABS(Mouse Y - CY)
X1 = CX + DeltaX,  Y1 = CY - DeltaY
X2 = CX + DeltaX,  Y2 = CY - DeltaY
X3 = CX + DeltaX,  Y3 = CY - DeltaY
X4 = CX + DeltaX,  Y4 = CY - DeltaY
```
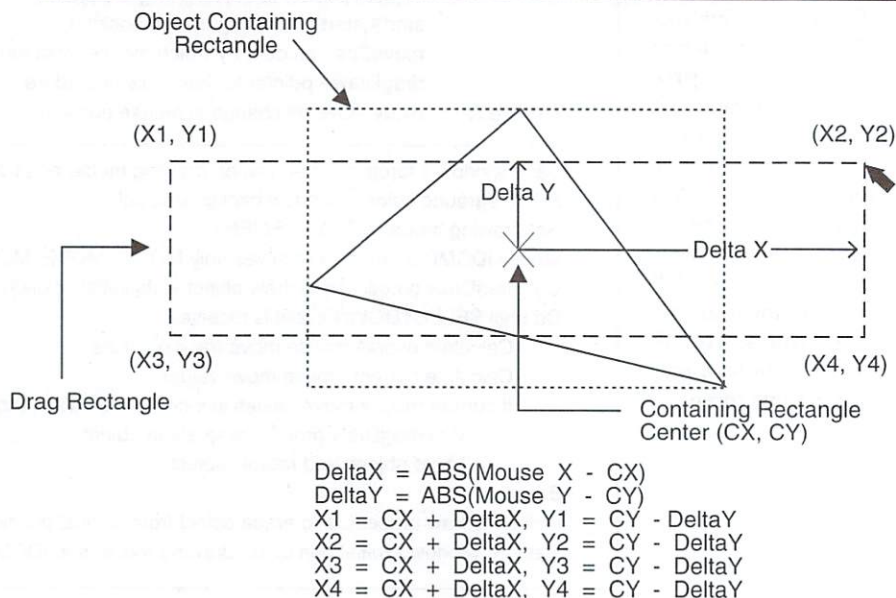
The main differences in the "action" procedures are the way the drag rectangles are set up and how the changes in the mouse position are used to calculate transform values. DragAndMove( ) and dragAndSize( ) construct the drag rectangle as an array of pixel coordinates. This is done by transforming the selected object's minimum containing rectangle world coordinates to view coordinates. DragAndRotate( ) constructs the rectangle as a polygon array in world coordinates, and also saves the initial rotation angle in the polygon array. Both dragAndSize( ) and dragAndRotate( ) construct the drag rectangle by finding the center coordinates of the selected object's containing rectangle, then subtracting and adding half the rectangle's width and height to the center coordinates. This makes the center of the rectangle its drag point, which simplifies the sizing and rotation calculations. DragZoomBox( ) constructs a pixel coordinate rectangle whose left, top and right, bottom coordinates are the same. As the mouse cursor moves, the rectangle "grows" in the direction the mouse is moving.

After handleDrag( ) returns, dragAndMove( ) finds x and y translation values in world units. To find the translation values, the change in pixel coordinate values are added to the pixel coordinates corresponding to the world center coordinates. The resulting view coordinates are then transformed to world coordinates, which yields the directed world distances along the x and y axes. DragAndMove( ) sends the translation values to moveObject( ), which adds the values to the object's coordinates.

For sizeAndDrag( ) and rotateAndDrag( ), the data needed to find the scale and rotation transform values are obtained by comparing the rectangle's initial values with its final values. SizeAndDrag( ) finds x and y scale values for resizing the object by simply calculating the ratio between the original width and height of the rectangle and its width and height after the drag is complete. Pixel values are adequate for finding scale values. The object is resized in resizeObject( ), which creates a transform matrix, then scales the object by transforming each of its coordinates. RotateAndDrag( ) computes the change in the object's rotation angle by subtracting the drag start angle from the final rotation angle. The change in the object's rotation angle is then sent to rotateObject( ). Like resizeObject( ), rotateObject( ) creates a transform matrix and rotates the object by transforming each of its coordinates.

After handleDrag( ) returns to dragZoomBox( ), the coordinates of the newly-created rectangle are used to find view transform values which will fit the world area contained in the rectangle into the Intuition window. This is done by converting the pixel rectangle coordinates to world coordinates, then calling worldWindowToViewWindow( ) to find the translation and scale values for the new view transform.

Although obtaining values through "pick and drag" interaction involves quite a bit of detailed code, the resulting programs are elegant and easy to use. Further, all drag interactions use the same basic techniques as those just discussed and presented in the demonstration program. If you understand how

Object Containing Rectangle

DY

DX

Containing Rectangle Center (CX, CY)

Drag Rectangle

DX = Mouse X - CX
DY = Mouse Y - CY
Rotation Angle = atan2 (-DY, DX)
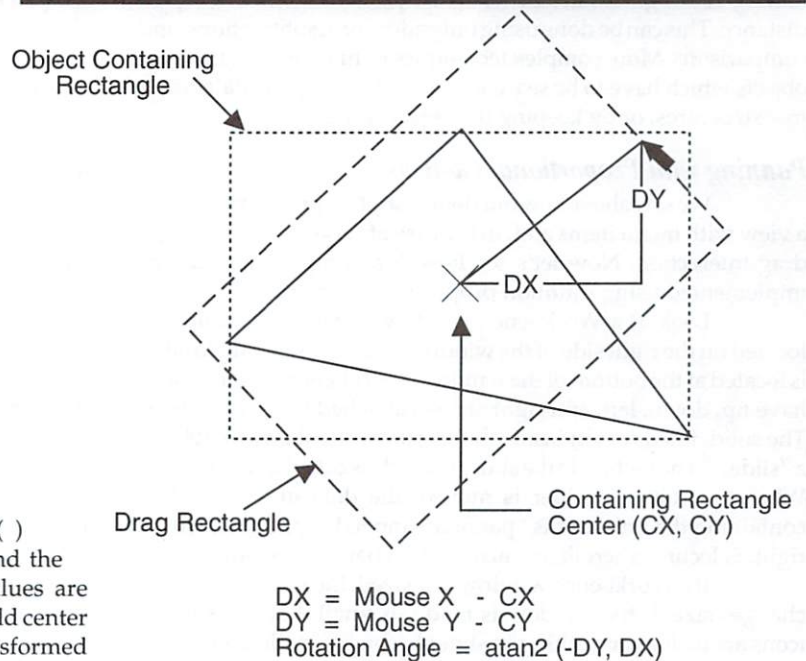
handleDrag( ) works, and how it interacts with its caller and with its "dragDraw" procedures, you should be able to easily modify the procedures and implement various types of drag interaction in your own programs.

Selecting Objects

Before an object can be dragged, it has to be selected. When a select button "pick" event occurs, how do we determine if an object was selected? What we have to do is find the object which is closest to the pick coordinates, then decide if it is "close enough" to the coordinates to qualify as being selected. In the demonstration program, "pick events" are handled in windowEvent( ), which calls findObject( ) to determine which world object is closest to the pick coordinates. If the nearest object is within three pixels of the pick coordinates, the object is selected. Before calling findObject( ), windowEvent( ) transforms the pick coordinates to world coordinates, and finds world coordinate distances along the x and y axes corresponding to three pixels. The larger of these distances is used as the "close enough" pick value. FindObject( ) iterates on the list of world objects and calls point2Poly( ) to actually calculate the distance from the pick coordinates to each object. If an object within the pick distance is found, findObject( ) returns the pointer to the object. Point2Poly( ) finds the minimum distance from a point to a polygon by calculating the minimum distance from the point to each of the line segments in the polygon.

The demonstration program doesn't "screen" objects before calculating the distance from the pick point to each object. It just examines every object until it finds one which is within the pick distance of the pick point. In actual CAD programs where

→

hundreds or thousands of objects are present, a variety of techniques are used to avoid calculating distances. An effective and simple technique is to screen out objects whose minimum containing rectangle is farther from the pick point than the pick distance. This can be done using only additions, subtractions, and comparisons. More complex techniques minimize the number of objects which have to be searched by partitioning the data using tree structures, or by keeping the objects sorted.

Panning with Proportional Gadgets

We saw above how our demonstration program "zooms" a view with menu items and, in the case of "zoom-box," through drag interaction. Now let's see how "panning" a view can be implemented using Intuition proportional gadgets.

Look at a Workbench window. A vertical scrollbar is located on the right side of the window and a horizontal scrollbar is located at the bottom of the window. Workbench windows also have up, down, left, and right arrows attached to the scrollbars. The solid, filled region inside the scrollbars is called a "knob" or a "slider." The rectangle the slider is inside is called a "container." When a scrollbar's slider is moved, the data in the window containing the scrollbar is "panned" (moved) up, down, left, or right. Selecting a scrollbar's arrows also pans the window.

If a Workbench window is resized, the scrollbar's slider changes size. If the window is made so small that some of the icons are no longer visible, the slider becomes smaller. Similarly, if the window is resized so that all of its icons are visible, the slider becomes larger and completely fills its container. Scrollbars are designed to provide a visual indication of the amount of data in a window which is visible, and also indicate where the "hidden"

data is located with respect to the window. The size of the slider reflects the percentage of the window's data which is visible. The size of the unfilled regions inside the slider's container represents the percentage of the window's data which is to the left, right, above, or below the window.

Let's see how to create scrollbars for our drawing window and hook them up to our view transform procedures. We'll use Intuition's proportional gadget and let Intuition take care of moving the slider when a user drags it, or clicks the select button in the slider's container. To keep life simple, we'll omit the scrollbar arrows. To create a proportional gadget, we need to set up three data structures. They are a Gadget structure, a PropInfo structure, and an Image structure. The Image structure contains the "imagery" for the slider. We can create our own slider image, or let Intuition create one for us. We'll take the easy path and let Intuition create our slider. We still have to provide Intuition with an Image structure it can use for the slider, but we don't have to initialize it.

The PropInfo structure contains all the interesting information about proportional gadgets. This structure is used to set up the initial size and location of the slider. Later, when the slider is moved, Intuition updates the values in the PropInfo structure and our program can look at the values to tell how much to pan our view window. If the program's Intuition window is resized or if the size of our model world changes, our program can also update the values in the structure so the slider's position and size reflects the new window or world size. The PropInfo structure members we are interested in are the "Flags" member, the "HorizPot" and "VertPot" members, and the "HorizBody" and "VertBody" members. Intuition uses the rest of the PropInfo members and we don't need to worry about them.

The Flags member contains the bit values which let Intuition know whether we want it to automatically take care of creating and moving the slider, and know the direction in which the slider can be moved. For the vertical scrollbar, we set Flags to AUTOKNOB | FREEVERT. The Flags value for the horizontal scrollbar is set to AUTOKNOB | FREEHORIZ. AUTOKNOB tells Intuition to take care of the slider. FREEVERT and FREEHORIZ specify the direction the slider can be moved: FREEVERT for the vertical slider, and FREEHORIZ for the horizontal slider.

The Pot variables and Body variables represent the percentage of the data in a window which is hidden (Pot) and visible (Body). The Body variables are used by Intuition to determine the size of the slider and how much to move it when the select button is clicked in an empty part of the slider's container. Both the Pot and Body variables are "unsigned short" and range in value from 0 to 65537. Intuition defines macros called MAXPOT and MAXBODY which can be used for the largest Pot and Body values. To use the Pot and Body values, we have to interpret them as percentages: 0 is interpreted as 0% (none), and MAXPOT and MAXBODY are interpreted as (al-

Figure Four
How the POT and BODY Values Are Related to the Parts of Our Model World.

World Window

Leftmost 25% of world window hidden

View Window

50% of world window width visible in the view window

Rightmost 25% of world window hidden

Container

Scrollbar

Slider

Leftmost 25% of container is empty

Rightmost 25% of container is empty

slider width = 50% of container width
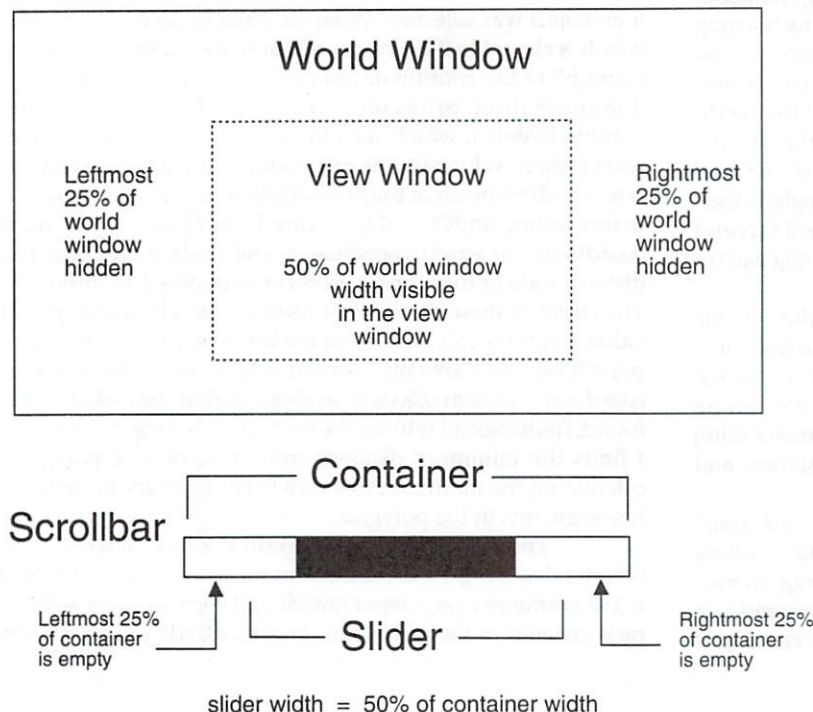
most) 100% (all). When Pot is 0, all of our model world is visible in the drawing window, and when it is MAXPOT, none of our model world is visible. Likewise, when Body is MAXBODY, all of our model world is visible and the slider fills the entire container. The Body value in our program can never be 0, since some of our model world will always be visible. The Pot value is used by Intuition to determine where to locate the left side of a horizontal slider and the top of a vertical slider.

To see how the Pot and Body values are related to each other and to the data which is displayed in an Intuition window, suppose 50% of our model world is visible and 50% of it is hidden. The size of the slider will then be 50% of the size of its container, and 50% of the container will be empty. If 25% of the model world is to the left of our Intuition window, the leftmost 25% and the rightmost 25% of the container will be empty. Figure 4 shows how the Pot and Body values are related to the visible and hidden parts of our model world.

Let's develop the formulas we'll need to set the Pot and Body values and to determine what the values mean when our program receives "pan" events. We'll only look at the horizontal scrollbar, since the calculations are similar for the vertical scrollbar.

To set the Body value, we need to determine the percentage of the model world which is visible in the Intuition window, and to set the Pot value, we need to determine the percentage of the model world which is hidden to the left of the window. The first step is to find the left and right coordinates of our model world in view coordinates. To do this, we transform the world minx and maxx values, using point2dForward( ). We then find the left and right coordinates of our view window. After we have these values, we can calculate the percentages. If wl and wr are the left and right coordinates of the world (in view coordinates), and vl and vr are the left and right coordinates of the view window, then the formulas are:

```
world width:                       wwidth = wr - wl
data hidden on the left:   lhid = MAX(vl - wl,0)
data hidden on the right: rhid = MAX(wr - vr,0)
total hidden data:            hidden = lhid + rhid
percent of world visible: visPercent = (wwidth-hidden)/wwidth
percent of world hidden on the left: leftPercent = MAX(lhid,0)/hidden

Body value = visPercent * MAXBODY
Pot value = leftPercent * MAXPOT
```

Why do we use MAX to find the hidden data values? Because, if the view is zoomed out far enough, the left and right sides of the world will be inside the view window and the values will be negative. In this case, the hidden percentage should be 0. Using the MAX macro takes care of this situation.

The above formulas are used in setPanGadgets( ) to set the Body and Pot values. After the Body and Pot values are found, NewModifyProp( ) is called to let Intuition know that the scrollbars are to be updated. SetPanGadgets( ) is called any time the size of the world or the size of the view window changes.

When a scrollbar slider is moved, our program receives a GADGETUP event, and the input handler calls panView( ) to handle the event. To pan our view window, we need to find new translation values, update our view transform, and erase and

redraw the model world. We can find the x translation value which will pan the view from two values: the number of world coordinate units which are actually hidden to the left of the window, and the number of units which will be hidden after the view is panned. If the number of units which are to be hidden is subtracted from the number of units currently hidden, the result will be the change in the current translation value which will pan the view. We saw above how to find the number of hidden units to the left of the window. To find how many units will be hidden after the view is panned, we multiply the total number of hidden units times the percentage represented by the Pot value. The first step in the calculations is to transform the world extent to view coordinates. We then use the following formulas:

```
data hidden on the left:   lhid =  MAX(vl - wl,0)
data hidden on the right: rhid =  MAX(wr - vr,0)
total hidden data:            hidden = lhid + rhid
percent of data to hide on the left: leftPercent = Pot / MAXPOT
world units to hide on the left: hide = leftPercent * hidden
x translation increment:  txInc = lhid - hide
x translation value: tx = current x translation value + txInc
```

The value calculated for txInc is an incremental translation value. By adding the increment to the current translation value, we arrive at the actual translation value for our new view transform.

→

There is one case where these translation formulas won't work: if the entire world is completely inside the view window. As noted above, this happens if we zoom the view way out. We can detect this situation by looking at the Body value: if it is equal to MAXBODY, the entire world is inside the view window. Even if this is the case, we may still be able to pan our view to center the world in the view window. To do this, we find the translation increment by subtracting the center coordinate of the world window from the center coordinate of the view window. The formulas are:

```
world center x:   wc = (wl + wr)/2
view center x:    vc = (vl + vr)/2
x translation increment: txInc = vc - wc
x translation value: tx = current x translation value + txInc
```

After we have our translation values, we erase everything, update the view transform, and then redisplay the model world. Before calling it quits, there is one more detail we need to take care of: panning may bring the entire model world into view. This happens when the view is first zoomed out and then panned. When this occurs, we need to call NewModifyProp( ) to reset Pot to 0 and Body to MAXBODY.

Now, for the easy part. When we create our scrollbars, we use relative positioning and relative sizing for them. This is done by using negative values for the scrollbar location and width (or height), and by setting the "GREL" flags in the gadget structure's Flags member. For our horizontal scrollbar, we use GRELWIDTH and GRELBOTTOM to specify that the width of the scrollbar is relative to the width of the window, and its location is relative to the bottom of the window. GRELHEIGHT and GRELRIGHT are used for the vertical scrollbar. By using relative positioning and sizing, we let Intuition do all the work of keeping the scrollbars the right size and in the right place.

Finally, to connect our pan event handler code, panView( ), to the scrollbar gadgets, we create an intuiExtension_t structure containing a pointer to panView( ). We then place the structure pointer in the UserData member of the scrollbar gadget structures, and set the RELVERIFY bit in the IDCMPFlags member. RELVERIFY tells Intuition to send us a GADGETUP event when the scrollbar is selected. When handleInput( ) receives a GADGETUP event, it retrieves the pointer to our event handler and calls it.

Creating and Running the Demonstration Program

To build an executable copy of the demonstration program, you will need to compile and link the demonstration program with the transform procedures. The source code for the demonstration program is in the file named panzoom.c, and the transform procedures are in the source file named transform.c. The Lattice (SAS) command to compile and link the program is

```
lc -Lm panzoom.c transform.c
```

The executable program can also be compiled and linked using the "make" file named panzoom.make. The command is

```
lmk -f panzoom.make
```

Either command will produce an executable program called "panzoom."

The program displays two polygons (a triangle and a rectangle) centered in an Intuition window. You can move, resize, or rotate the polygons by selecting an action from the "Action" menu, picking the polygon, then dragging it with the mouse. To complete the action and transform the polygon, click the select button. If the polygons are moved or resized so that they no longer fit in the Intuition window, selecting the "View-All" menu item will calculate a new transform to fit the model world in the window and redisplay the polygons. The view can be zoomed in or out by a factor of 2 by picking either "View-Zoom-In" or "View-Zoom-Out". You can also zoom in on a region by first picking the "View-Zoom-Box" menu item, then drawing a zoom rectangle in the Intuition window. The zoom box is drawn by first selecting a corner of the zoom box, then dragging the mouse to the opposite corner. The rectangle corners are selected by clicking the select button while the cursor is over the desired location. The model world can be panned by picking the scrollbars in the bottom or the right side of the window. The program is ended by picking the window's "close" gadget or by selecting the "project-quit" menu item.

Summary and Preview

In this article, we've looked at event-driven programming techniques and "drag" interaction. We've seen how these techniques can be combined with coordinate transforms in vector-based CAD programs to modify graphical objects and pan and zoom drawings. We've now developed most of the basic building blocks of a vector-based CAD program: transforms and general-purpose input and drag handlers. The building blocks that are missing are geometric objects we can use to model world objects: lines, circles, rectangles, etc. Next time, we'll conclude this series by using object-oriented programming techniques (honest!) to create and manage geometric objects. Until then, if you want to experiment with the techniques presented in this article, add an "Insert-rectangle" menu item and code to create rectangles. Hint: look at how "zoom-box" works. You can also add a grid to the program to experiment with transforms.

```
/*=========================================================*/
/*
 *   Listing 1.  panzoom.c - program to demonstrate/test
 *               world & view transform procedures,
 *               pan and zoom, and event-driven programming
 *               techniques.
 *
 *   copyright (c) 1991 by Forest W. Arnold
 */
/*=========================================================*/

#include <stdio.h>
#include <exec/types.h>
#include <intuition/intuition.h>
#include <graphics/gfxbase.h>
#include <graphics/regions.h>
#include <graphics/clip.h>
#include <graphics/layers.h>
#include "transform.h"

/*  intuition stuff                                       */

#define INTUITION_REV   34
#define GRAPHICS_REV    34
extern struct IntuitionBase *IntuitionBase;
extern struct GfxBase       *GfxBase;
struct LayersBase           *LayersBase;

/*  event handler procedure and data typedefs             */

typedef int (*eventProc_t)( struct IntuiMessage *,void * );
typedef int (*actionProc_t)( struct Window *,long,void * );

typedef struct _intuiExtension    /* Intuition extension  */
{
    eventProc_t handler;        /* event handler          */
    void        *data;          /* data to be sent        */
} intuiExtension_t;

typedef struct _miData            /* menu item data       */
{
    actionProc_t   action;      /* menu item action proc  */
    long           modifier;    /* action modifier        */
    void           *data;       /* any data to be sent    */
} miData_t;

typedef struct _mItem             /* extended menu item    */
{
    struct MenuItem   mi;       /* intuition menu item    */
    intuiExtension_t  extens;   /* menu item extension    */
} myMenuItem_t;

/*  define an object type                                 */

typedef struct _object
{
    struct _object *next;       /* list link field        */
    double         minx,miny;   /* lower,left corner       */
    double         maxx,maxy;   /* upper,right corner      */
    double         *poly;       /* object's shape          */
    int            npts;        /* # of points for shape   */
} object_t;

/*
 *  set up some polygons to display.  The coordinates are
 *  stored in the arrays as x0,y0,x1,y1...xn,yn.
 */

double triangle[] =
{ 400.0,-400.0, 200.0,0.0, 0.0,-400.0,400.0,-400.0 };
double square[] =
{ -500.0,100.0, -500.0,500.0, -100.0,500.0,
  -100.0,100.0, -500.0,100.0
};

object_t obj2 =
{ NULL,    0.0,-400.0, 400.0, 0.0,triangle,8 };
object_t obj1 =
{ &obj2,-500.0, 100.0,-100.0,500.0,square, 10 };

/*
 *  define and initialize a structure to hold the
 *  program state and the world data.
 */

typedef struct _world
{
    double      minx,miny;  /* world lower left coords    */
    double      maxx,maxy;  /* world upper right coords    */
    actionProc_t action;    /* current action procedure   */
    object_t    *selected;  /* active object              */
    void        *data;      /* arg for action procedure    */
    long        modifier;   /* arg for action procedure    */
    long        pickX,pickY;/* pixel pick coordinates      */
    double      wrldX,wrldY;/* world pick coordinates       */
```

```
    double      eps;        /* search epsilon value        */
    object_t    *objects;   /* list of world shapes        */
    transform2_t *viewTf;   /* window's view transform     */
    double      aspect;     /* window aspect ratio         */
} world_t;

world_t world =
{ -550.0,-550.0,550.0,550.0,NULL,NULL,NULL,
        0,0,0,0.0,0.0,0.0,&obj1,NULL,1.0
};

/*  Event handler procedures                              */

int windowEvent( struct IntuiMessage *,void * );
int miQuit( struct IntuiMessage *,void * );
int miSetAction( struct IntuiMessage *,void * );
int miFullView( struct IntuiMessage *,void * );
int miZoom( struct IntuiMessage *,void * );
int dragAndSize( struct Window *,long,void * );
int dragAndRotate( struct Window *,long,void * );
int dragAndMove( struct Window *,long,void * );
int dragZoomBox( struct Window *,long,void * );

/*  Menu text definitions                                 */

struct IntuiText projTxt[] =      /* project text          */
{
    { 2,1,JAM2,2,1,NULL,"Quit",  NULL },
};
struct IntuiText actionTxt[] =    /* action text           */
{
    { 2,1,JAM2,2,1,NULL,"Move",  NULL },
    { 2,1,JAM2,2,1,NULL,"Size",  NULL },
    { 2,1,JAM2,2,1,NULL,"Rotate",NULL },
};
struct IntuiText viewTxt[] =      /* view text             */
{
    { 2,1,JAM2,2,1,NULL,"All",    NULL },
    { 2,1,JAM2,2,1,NULL,"Zoom ->",NULL },
};
struct IntuiText zoomTxt[] =      /* zoom text             */
{
    { 2,1,JAM2,2,1,NULL,"In",  NULL },
    { 2,1,JAM2,2,1,NULL,"Out", NULL },
    { 2,1,JAM2,2,1,NULL,"Box", NULL },
};

/*  menu modifiers (stored with subitem)                  */

#define ZOOMIN    1
#define ZOOMOUT   2

/*  menu item extension structure data                    */

miData_t moveData =     { dragAndMove,    0,      NULL };
miData_t sizeData =     { dragAndSize,    0,      NULL };
miData_t rotateData =   { dragAndRotate,  0,      NULL };
miData_t zoomInData =   { NULL,           ZOOMIN, NULL };
miData_t zoomOutData =  { NULL,           ZOOMOUT,NULL };
miData_t zoomBoxData =  { dragZoomBox,    0,      NULL };

/*  menu subitem definitions                              */

myMenuItem_t zoomItm[] =
{
    { /* "Zoom In" sub-item      */
      { &(zoomItm[1].mi),0,0,0,0,
        ITEMTEXT|ITEMENABLED|HIGHCOMP,0,
        (APTR)&zoomTxt[0],NULL,NULL,NULL,MENUNULL },
      { miZoom,(void*)&zoomInData }
    },
    { /* "Zoom Out" sub-item     */
      { &(zoomItm[2].mi),0,0,0,0,
        ITEMTEXT|ITEMENABLED|HIGHCOMP,0,
        (APTR)&zoomTxt[1],NULL,NULL,NULL,MENUNULL },
      { miZoom,(void*)&zoomOutData }
    },
    { /* "Zoom Box" sub-item     */
      { NULL,          0,0,0,0,
        ITEMTEXT|ITEMENABLED|HIGHCOMP,0,
        (APTR)&zoomTxt[2],NULL,NULL,NULL,MENUNULL },
      { miSetAction,(void*)&zoomBoxData }
    },
};

/*  menu item definitions                                 */

myMenuItem_t projItm[] =
{
    { /* "Quit" menuitem         */
      { NULL,0,0,0,0,
        ITEMTEXT|ITEMENABLED|HIGHCOMP,0,
        (APTR)&projTxt[0],NULL,NULL,NULL,MENUNULL },
      { miQuit,NULL  }
    },
};
myMenuItem_t actionItm[] =
{
    { /* "Move" menuitem         */
```

```
        {  &(actionItm[1].mi),0,0,0,0,
           ITEMTEXT|ITEMENABLED|HIGHCOMP,0,
           (APTR)&actionTxt[0],NULL,NULL,NULL,MENUNULL },
        {  miSetAction,(void*)&moveData  }
    },
    {  /* "Size" menuitem          */
        {  &(actionItm[2].mi),0,0,0,0,
           ITEMTEXT|ITEMENABLED|HIGHCOMP,0,
           (APTR)&actionTxt[1],NULL,NULL,NULL,MENUNULL },
        {  miSetAction,(void*)&sizeData }
    },
    {  /* "Rotate" menuitem        */
        {  NULL,       0,0,0,0,
           ITEMTEXT|ITEMENABLED|HIGHCOMP,0,
           (APTR)&actionTxt[2],NULL,NULL,NULL,MENUNULL },
        {  miSetAction,(void*)&rotateData }
    },
};
myMenuItem_t viewItm[] =
{
    {  /* "All" menuitem          */
        {  &(viewItm[1].mi),0,0,0,0,
           ITEMTEXT|ITEMENABLED|HIGHCOMP,0,
           (APTR)&viewTxt[0],NULL,NULL,NULL,MENUNULL },
        {  miFullView,NULL }
    },
    {  /* "Zoom" menuitem         */
        {  NULL,0,0,0,0,
           ITEMTEXT|ITEMENABLED|HIGHCOMP,0,
           (APTR)&viewTxt[1],NULL,NULL,
           (struct MenuItem *)&zoomItm[0],MENUNULL),
        {  NULL,NULL }
    },
};

/*  These are the Menu definitions               */

struct Menu menu[] =
{
    {  /* "Project" menu              */
       &menu[1],0,0,0,0,MENUENABLED,
       "Project",(struct MenuItem*)&projItm[0]
    },
    {  /* "Action" menu              */
       &menu[2],0,0,0,0,MENUENABLED,
       "Action",(struct MenuItem*)&actionItm[0]
    },
    {  /* "View  " menu              */
       NULL,0,0,0,0,MENUENABLED,
       "View ",(struct MenuItem*)&viewItm[0]
    },
};

/*  scrollbar (pan) gadget definitions           */

#define PANHORIZ  1
#define PANVERT   2

/*  pan event handler and extension data          */

int panView( struct IntuiMessage *msg,void * );
intuiExtension_t panExt = { panView,NULL  };

struct Image hImage;   /*  reserve space for intuition */
struct Image vImage;   /*  reserve space for intuition */

struct PropInfo hInfo =
{  AUTOKNOB|FREEHORIZ,0,0,-1,-1, };
struct PropInfo vInfo =
{  AUTOKNOB|FREEVERT, 0,0,-1,-1, };

struct Gadget hPan =
{
    NULL,1,-8,-15,9,GADGHCOMP|GRELWIDTH|GRELBOTTOM,
    RELVERIFY|BOTTOMBORDER,PROPGADGET,(APTR)&hImage,NULL,
    NULL,0,(APTR)&hInfo,PANHORIZ,(APTR)&panExt
};
struct Gadget vPan =
{
    &hPan,-15,10,16,-18,GADGHCOMP|GRELHEIGHT|GRELRIGHT,
    RELVERIFY|RIGHTBORDER,PROPGADGET,(APTR)&vImage,NULL,
    NULL,0,(APTR)&vInfo,PANVERT,(APTR)&panExt
};

/*  pen number definitions and drag code definitions   */

#define BGPEN      0
#define DRAWPEN    1
#define HLPEN      3
#define AXISPEN    2

#define BEGINDRAG 1
#define ENDDRAG   2

#define MIN(x,y) ( ((x) < (y)) ? (x) : (y) )
#define MAX(x,y) ( ((x) > (y)) ? (x) : (y) )

/*  forward procedure declarations               */
```

```
void handleInput( struct Window * );
void handleDrag( struct Window *,void *,long,long,long,
                 void (*)(),long *,long * );
void sizeAndLocateMenus( struct Menu *,short,short);
void sizeAndLocateMenuItems( struct MenuItem *,short,
                             short,short,short,short );
int maxMiTextLength( struct MenuItem * );
struct Window *displayWindow(int,int,int,int,
                             char *,struct Gadget * );
void    closeLibs( void );
int     openLibs( void );
void fullDisplay( struct Window * );
void zoomView( struct Window *,double );
void setPanGadgets( struct Window * );
object_t *findObject( object_t *,double,double,double );
double point2Poly( object_t *,double,double );
void drawAll( struct Window * );
void drawAxis( struct Window * );
void drawObject(struct Window *,transform2_t *,object_t *);
void eraseAll( struct Window * );
void moveObject( object_t *,double,double );
void resizeObject( object_t *,double,double);
void rotateObject( object_t *,double);
void moveDrag(struct Window *,void *,long,long,
              long,long,long);
void sizeDrag(struct Window *,void *,long,long,
              long,long,long);
void rotateDrag(struct Window *,void *,long,long,
                long,long,long);
void zoomDrag(struct Window *,void *,long,long,
              long,long,long);
void drawRect( struct RastPort *,long,long,long,long);
void setWorldExtent(world_t *);
void getViewExtent(struct Window *,
                   long *,long *,long *,long *);
struct Region *setClipRect( struct Window *,
                            long,long,long,long);
void removeClipRect(struct Window *,struct Region *);

/*=========================================================*/

void main( int argc,char** argv )
{
    intuiExtension_t windowExt =
                     { windowEvent,(void*)&world };
    struct Screen screen;
    struct Window *window = NULL;
    transform2_t  tForm;
    long          dpmX,dpmY;
    short         txtHeight,txtWidth;

    /*  open libraries & get screen info            */

    if ( ! openLibs() )
    {
        printf("\nCan't open system libraries...\n");
        exit(1);
    }
    GetScreenData((char*)&screen,sizeof(struct Screen),
                  WBENCHSCREEN,NULL);

    /*  calculate x to y aspect ratio & text dimensions   */

    dpmX = (long)GfxBase->NormalDPMX;
    dpmY = (long)GfxBase->NormalDPMY;
    if ( screen.Height == 200 )
        dpmY /= 2;
    world.aspect = (double)dpmX/(double)dpmY;

    txtHeight = screen.RastPort.Font->tf_YSize;
    txtWidth = screen.RastPort.Font->tf_XSize;

    /*  set up menu positions and sizes            */

    sizeAndLocateMenus( &menu[0],txtHeight,txtWidth );

    /*  open the window and attach menu            */

    window = displayWindow(50,50,300,150,"Transforms",&vPan);
    if ( ! window )
    {
        printf("Can't open window\n");
        goto clean;
    }
    SetMenuStrip(window,&menu[0]);

    /*  set up transform & draw the world          */

    world.viewTf = &tForm;
    window->UserData = (UBYTE*)&windowExt;
    fullDisplay( window );

    /*  handle input until window closed            */

    handleInput( window );

clean:

    if ( window )
```

```
    {
        ClearMenuStrip( window );
        CloseWindow( window );
    }
    closeLibs();
    exit(0);
}
/*============================================================
 *  sizeAndLocateMenus() - Calculate positions and sizes
 *      for all menus in a menu strip.
 *============================================================*/

void sizeAndLocateMenus( struct Menu *menu,
                            short tHt,short tWd )
{
    SHORT start = 2;

    while( menu )
    {
        menu->LeftEdge = start;
        menu->Width = tWd * strlen(menu->MenuName) + tWd;

        if ( menu->FirstItem )
            sizeAndLocateMenuItems( menu->FirstItem,
                            menu->Width,0,0,tHt,tWd );
        start += menu->Width + 2;
        menu = menu->NextMenu;
    }
}
/*============================================================
 *  sizeAndLocateMenuItems() -  Calculate positions and
 *      sizes for all menu items and subitems in a menu
 *============================================================*/

void sizeAndLocateMenuItems( struct MenuItem *mi,
                            short minw,short l,short t,
                            short tHt,short tWd )
{
    int txtlen;

    txtlen = maxMiTextLength( mi );

    while( mi )
    {
        mi->TopEdge = t;
        mi->LeftEdge = l;
        mi->Width = tWd * txtlen + 4;

        if ( mi->Width < minw )
            mi->Width = minw;
        mi->Height = tHt + 2;

        if ( mi->SubItem )
            sizeAndLocateMenuItems( mi->SubItem,0,mi->Width-8,
                            0,tHt,tWd );
        t += mi->Height;
        mi = mi->NextItem;
    }
}
/*============================================================
 *  maxMiTextLength() - find longest menu item text
 *============================================================*/

int maxMiTextLength( struct MenuItem *mi )
{
    struct IntuiText *itext;
    int len,retlen;

    retlen = 0;

    while( mi )
    {
        itext = (struct IntuiText *)mi->ItemFill;
        len = strlen(itext->IText);
        retlen = len > retlen ? len : retlen;
        mi = mi->NextItem;
    }
    return retlen;
}
/*============================================================
 *  miQuit() - Menu item pick event handler.  Returns
 *      non-zero value to stop input processing.
 *============================================================*/

int miQuit( struct IntuiMessage *msg,void *data )
{
    return -1;   /* non-zero will end input loop       */
}
/*============================================================
 *  miSetAction() - Menu item pick event handler.  Sets
 *      world state info for the picked action.
 *============================================================*/

int miSetAction( struct IntuiMessage *msg,void *data )
{
    miData_t *itmData = (miData_t *)data;

    if ( itmData )
    {
```

```
        world.action = itmData->action;
        world.modifier = itmData->modifier;
        world.data = itmData->data;
    }
    else
    {
        world.action = NULL;
        world.modifier = 0;
        world.data = NULL;
    }
    return 0;
}
/*============================================================
 *  miFullView() - Menu item pick event handler.
 *      Redisplays full model world centered in window.
 *============================================================*/

int miFullView(struct IntuiMessage *msg,void *data )
{
    fullDisplay(msg->IDCMPWindow);
    return 0;
}
/*============================================================
 *  miZoom() - Menu item pick event handler.  Zooms the
 *      view window in or out by a factor of 2.
 *============================================================*/

int miZoom(struct IntuiMessage *msg,void *data )
{
    miData_t *itmData = (miData_t *)data;

    if ( itmData && itmData->modifier == ZOOMIN )
        zoomView(nsg->IDCMPWindow,2.0);
    else
        zoomView(nsg->IDCMPWindow,0.5);
    return 0;
}
/*============================================================
 *  windowEvent() - Window event handler.  Called when
 *      an event occurs inside the input window to take
 *      care of window-specific events, such as "close",
 *      "button press", etc.
 *============================================================*/

int windowEvent(struct IntuiMessage *msg,void *data )
{
    struct Window *window = msg->IDCMPWindow;
    double xz,yz,x,y;

    if ( msg->Class == CLOSEWINDOW )
        return 1;
    else
    if ( msg->Class == REFRESHWINDOW )
        drawAll(window);
    else
    if ( msg->Class == NEWSIZE )
        setPanGadgets(window);  /* update pan gadgets     */
    else
    if ( msg->Class == MOUSEBUTTONS &&
            msg->Code == SELECTDOWN )
    {
        /*
         * Calculate the search epsilon value
         * in world units corresponding to three pixels
         * and find the world coordinate pick values.
         */

        point2dForward( world.viewTf,0.0,0.0,&xz,&yz);
        point2dInverse( world.viewTf,xz+3.0,yz+3.0,&x,&y);
        x = ABS(x);   y = ABS(y);
        world.eps = MAX(x,y);
        point2dInverse( world.viewTf,(double)msg->MouseX,
                        (double)msg->MouseY,&x,&y );

        /*
         * if an object has not already been selected,
         * see if one is within the pick epsilon distance
         * of the pick coordinates
         */

        if ( world.selected == NULL )
            world.selected = findObject( world.objects,
                            x,y,world.eps );
        world.pickX = msg->MouseX;
        world.pickY = msg->MouseY;
        world.wrldX = x;
        world.wrldY = y;

        /*  If we have an action procedure, call it       */

        if ( world.action )
            (void)(*world.action)(window,
                            world.modifier,world.data);
        world.selected = NULL;
    }
    return 0;
}
```

```
/*========================================================
 * fullDisplay() - called to recalculate the view
 *    transform & redisplay the entire model world
 *    centered in the window.
 ========================================================*/

void fullDisplay( struct Window *window )
{
    double tx,ty,sx,sy;
    long vxl,vyb,vxr,vyt;

    getViewExtent(window,&vxl,&vyt,&vxr,&vyb);
    setWorldExtent( &world );
    eraseAll( window );
    worldWindowToViewWindow( world.minx,world.miny,
                             world.maxx,world.maxy,
                             (double)vxl,(double)vyb,
                             (double)vxr,(double)vyt,
                             world.aspect,&tx,&ty,&sx,&sy );
    (void)setViewTransform( world.viewTf,tx,ty,sx,sy,0.0 );
    SetAPen(window->RPort,DRAWPEN);
    drawAll( window );
    setPanGadgets(window);   /* update pan gadgets      */
}
/*========================================================
 * zoomView() - Modifies the view transform scale values
 *    and redisplays the view.
 ========================================================*/

void zoomView( struct Window *window,double scaleVal )
{
    double tx,ty,sx,sy,ang;

    /* get previous transform values and update them   */

    eraseAll( window );
    getTransform( world.viewTf,&tx,&ty,&sx,&sy,&ang );
    sx *= scaleVal;  sy *= scaleVal;
    (void)setViewTransform( world.viewTf,tx,ty,sx,sy,ang );
    SetAPen(window->RPort,DRAWPEN);
    drawAll( window );
    setPanGadgets(window);
}
/*========================================================
 * panView() - Pan gadget event handler.  Called when
 *    GADGETUP event generated for pan gadget.
 ========================================================*/

int panView( struct IntuiMessage *msg,void *data )
{
    struct Gadget *gadg = (struct Gadget*)msg->IAddress;
    struct Window *window = msg->IDCMPWindow;
    struct PropInfo *info;
    double xl,yb,xr,yt;        /* transformed world extent*/
    double tx,ty,sx,sy,ang;    /* transform values       */
    double oldTx,oldTy;        /* prev transform values  */
    double hidden,lHid,tHid;   /* 'hidden' amounts       */
    long l,b,r,t;              /* window extent          */
    long wl,wr,wt,wb;          /* world pixel extent      */
    long wc,c;                 /* world,window center    */
    long dx,dy;                /* change in coordinates  */

    /*
     * get window extent & world extent in window
     * coordinates.  Get existing transform parameters.
     */

    getViewExtent(window,&l,&t,&r,&b);
    point2dForward( world.viewTf,world.minx,world.miny,
                                 &xl,&yb);
    point2dForward( world.viewTf,world.maxx,world.maxy,
                                 &xr,&yt);
    wl = doubleToLong(xl);  wr = doubleToLong(xr);
    wt = doubleToLong(yt);  wb = doubleToLong(yb);
    getTransform(world.viewTf,&tx,&ty,&sx,&sy,&ang);
    oldTx = tx;  oldTy = ty;

    /*
     * pot/MAXPOT represent the percentage of the hidden
     * information which is to the left of or above the
     * knob.  lHid is the number of world units to the
     * left of the window, and tHid is the number of
     * world units above the window.  Subtracting these
     * values from the previous left, top hidden values
     * ( l-wl & t-wt ) gives the change in the number of
     * world units.  If nothing is hidden, we just center
     * the world by moving the world center to the window
     * center.
     */

    info = (struct PropInfo*)gadg->SpecialInfo;

    if ( gadg->GadgetID == PANHORIZ )
    {
        if ( info->HorizBody != MAXBODY )
        {
            hidden = MAX(l-wl,0) + MAX(wr-r,0);
            lHid = info->HorizPot*hidden/MAXPOT;
            tx += (double)(l-wl) - lHid;
        }
```

```
        else
        {
            c = (r+l)/2;  wc = doubleToLong((xr+xl)/2.0);
            tx += (double)(c-wc);
        }
    }
    else
    {
        if ( info->VertBody != MAXBODY )
        {
            hidden = MAX(t-wt,0) + MAX(wb-b,0);
            tHid = info->VertPot*hidden/MAXPOT;
            ty += (double)(t-wt) - tHid;
        }
        else
        {
            c = (b+t)/2;  wc = doubleToLong((yb+yt)/2.0);
            ty += (double)(c-wc);
        }
    }

    /*  if nothing changed, just return                  */

    if ( oldTx == tx && oldTy == ty )
        return 0;

    /*  erase world, update view transform, redisplay    */

    eraseAll( window );
    (void)setViewTransform( world.viewTf,tx,ty,sx,sy,ang );
    SetAPen(window->RPort,DRAWPEN);
    drawAll( window );

    /*  if entire world is in view, reset pan gadgets    */

    dx = doubleToLong(tx-oldTx);
    if ( info->HorizBody != MAXBODY &&
         l <= wl + dx && wr + dx <= r )
        NewModifyProp(&hPan,window,NULL,AUTOKNOB|FREEHORIZ,
                      0,0,MAXBODY,-1,1);

    dy = doubleToLong(ty-oldTy);
    if ( info->VertBody != MAXBODY &&
         t <= wt+dy && wb+dy <= b )
        NewModifyProp(&vPan,window,NULL,AUTOKNOB|FREEVERT,
                      0,0,-1,MAXBODY,1);
    return 0;
}
/*========================================================
 * setPanGadgets() - calculate pan gadget sizes and
 *    positions when the view window changes.
 ========================================================*/

void setPanGadgets( struct Window *window )
{
    double xl,yb,xr,yt;        /* transformed world extents */
    long hHidden,vHidden;      /* hidden amounts            */
    long lHid,tHid;            /* hidden left,top amounts   */
    long width,height;         /* world pixel dimensions    */
    long l,b,r,t;              /* window extents            */
    long wl,wb,wr,wt;          /* world pixel extents       */
    unsigned short hBody,vBody,hPos,vPos;

    /*
     * get the window extent and the world extent in
     * window coordinates.  Find the total width &
     * height of the world.
     */

    getViewExtent(window,&l,&t,&r,&b);
    point2dForward( world.viewTf,world.minx,world.miny,
                                 &xl,&yb);
    point2dForward( world.viewTf,world.maxx,world.maxy,
                                 &xr,&yt);
    wl = doubleToLong(xl);  wr = doubleToLong(xr);
    wt = doubleToLong(yt);  wb = doubleToLong(yb);

    width = ABS(wr - wl);
    height = ABS(wb - wt);

    /*
     * (width - hHidden)/width gives the percentage
     * of the world width which is visible.  Multiply
     * this value times the max knob size to set the
     * knob size.
     */

    lHid = l - wl;
    hHidden = MAX(lHid,0) + MAX(wr-r,0);
    if ( hHidden > 0 )
        hBody = (width - hHidden)*MAXBODY/width;
    else
        hBody = MAXBODY;

    tHid = t - wt;
    vHidden = MAX(tHid,0) + MAX(wb-b,0);
    if ( vHidden > 0 )
        vBody = (height - vHidden)*MAXBODY/height;
    else
        vBody = MAXBODY;
```

```c
/*
 *  lHid/hHidden is the percentage of the hidden
 *  world units to the left of the window.  Multiply
 *  this value times the max pot size to set the
 *  space to the left side of the knob.
 */

if ( lHid > 0 )
    hPos = lHid*MAXPOT/hHidden;
else
    hPos = 0;

if ( tHid > 0 )
    vPos = tHid*MAXPOT/vHidden;
else
    vPos = 0;

NewModifyProp(&hPan,window,NULL,AUTOKNOB|FREEHORIZ,
              (ULONG)hPos,0,(ULONG)hBody,-1,1);
NewModifyProp(&vPan,window,NULL,AUTOKNOB|FREEVERT,
              0,(ULONG)vPos,-1,(ULONG)vBody,1);
}
/*========================================================
 *  dragZoomBox() - process zoom box creation
 ========================================================*/

int dragZoomBox(struct Window *window,long mod,void *data)
{
    struct Region   *prevRegion;
    double          minx,miny,maxx,maxy;
    double          xl,yb,xr,yt;
    double          sx,sy,tx,ty;
    long            rect[4];
    long            dx,dy,l,t,r,b;

    /*  clip to the view window rectangle      */

    getViewExtent(window,&l,&t,&r,&b);
    prevRegion = setClipRect(window,l,t,r,b);

    /*
     *  set up the pixel coordinate rectangle and call the
     *  drag handler.
     */

    rect[0] = rect[2] = world.pickX;
    rect[1] = rect[3] = world.pickY;

    handleDrag( window, (void*)&rect,world.pickX,
                world.pickY,1,zoomDrag,&dx,&dy );

    /*
     *  get the box dimensions and reset the transform,
     *  unless zoom box or scale values are too small
     */

    if ( ABS(dx) >= 4 && ABS(dy) >= 4 )
    {
        xl = (double)MIN(rect[0],rect[2]);
        yb = (double)MAX(rect[1],rect[3]);
        xr = (double)MAX(rect[0],rect[2]);
        yt = (double)MIN(rect[1],rect[3]);
        point2dInverse( world.viewTf,xl,yb,&minx,&miny );
        point2dInverse( world.viewTf,xr,yt,&maxx,&maxy );

        if ( ABS(maxx-minx) < 5.0 ||
             ABS(maxy-miny) < 5.0   )
            goto done;

        worldWindowToViewWindow( minx,miny,maxx,maxy,
                                 (double)l,(double)b,
                                 (double)r,(double)t,
                                 world.aspect,&tx,&ty,
                                 &sx,&sy);
        eraseAll( window );
        (void)setViewTransform(world.viewTf,tx,ty,sx,sy,0.0);
        SetAPen(window->RPort,DRAWPEN);
        removeClipRect( window,prevRegion );
        drawAll( window );
        setPanGadgets(window);
        world.action = NULL;
        return 0;
    }
done:
    removeClipRect( window,prevRegion );
    return 0;
}
/*========================================================
 *  dragAndMove() - process move interaction
 ========================================================*/

int dragAndMove(struct Window *window,long mod,void *data)
{
    struct Region   *prevRegion;
    object_t        *obj = world.selected;
    long            rect[4];
    double          xz,yz,x,y;
    long            dx,dy,l,t,r,b;
```

```c
    /*  If no object selected, just return     */

    if ( obj == NULL )
        return 0;

    /*
     *  clip to the view window rectangle and highlight
     *  the object
     */

    getViewExtent(window,&l,&t,&r,&b);
    prevRegion = setClipRect(window,l,t,r,b);
    SetAPen(window->RPort,HLPEN);
    drawObject(window,world.viewTf,obj);

    /*
     *  set up the pixel coordinate rectangle and call the
     *  drag handler.
     */

    point2dForward(world.viewTf,obj->minx,obj->miny,&x,&y);
    rect[0] = doubleToLong(x);   rect[1] = doubleToLong(y);
    point2dForward(world.viewTf,obj->maxx,obj->maxy,&x,&y);
    rect[2] = doubleToLong(x);   rect[3] = doubleToLong(y);

    handleDrag( window, (void*)&rect,world.pickX,
                world.pickY,1,moveDrag,&dx,&dy );

    /*
     *  erase object, find world distance corresponding
     *  to drag distance, and move the object.
     */

    SetAPen(window->RPort,BGPEN);
    drawObject(window,world.viewTf,obj);
    point2dForward( world.viewTf,0.0,0.0,&xz,&yz);
    point2dInverse( world.viewTf,
                    xz+(double)dx,yz+(double)dy,&x,&y);
    moveObject(obj,x,y);

    /*  remove the clip region & redraw everything      */

    removeClipRect( window,prevRegion );
    drawAll(window);

    /*  find new world extent and reset pan gadgets      */

    setWorldExtent( &world );
    setPanGadgets( window );
    return 0;
}
/*========================================================
 *  dragAndSize() - process size interaction
 ========================================================*/

int dragAndSize(struct Window *window,long mod,void *data)
{
    struct Region   *prevRegion;
    object_t        *obj = world.selected;
    double          sx,sy;
    double          num1,num2;
    double          x1,y1,x2,y2;
    long            rect[6];
    long            minx,miny,maxx,maxy;
    long            cx,cy,dx,dy;
    long            l,t,r,b;

    /*  If no object selected, just return     */

    if ( obj == NULL )
        return 0;

    /*
     *  clip to the view window rectangle and highlight
     *  the object
     */

    getViewExtent(window,&l,&t,&r,&b);
    prevRegion = setClipRect(window,l,t,r,b);
    SetAPen(window->RPort,HLPEN);
    drawObject(window,world.viewTf,obj);

    /*
     *  set up the pixel coordinate rectangle and draw it,
     *  then call the drag handler.
     */

    point2dForward(world.viewTf,obj->minx,obj->miny,&x1,&y1);
    point2dForward(world.viewTf,obj->maxx,obj->maxy,&x2,&y2);
    rect[0] = minx = doubleToLong(x1);
    rect[1] = miny = doubleToLong(y1);
    rect[2] = maxx = doubleToLong(x2);
    rect[3] = maxy = doubleToLong(y2);
    rect[4] = cx = (minx + maxx)/2;
    rect[5] = cy = (miny + maxy)/2;
    drawRect(window->RPort,minx,miny,maxx,maxy);

    handleDrag( window, (void*)&rect,cx,cy,1,
                sizeDrag,&dx,&dy );
```

```
      /*
       *  erase object and rectangle, find new scale values
       *  and resize the object.
       */

      SetAPen(window->RPort,BGPEN);
      drawObject(window,world.viewTf,obj);
      drawRect(window->RPort,minx,miny,maxx,maxy);

      num1 = ABS(rect[2]-rect[0]);
      if ( num1 < 1.0 )
         num1 = 1.0;
      num2 = ABS(rect[3]-rect[1]);
      if ( num2 < 1.0 )
         num2 = 1.0;
      sx = num1/(x2-x1);    sy = -num2/(y2-y1);
      resizeObject(obj,sx,sy);

      /*  remove the clip region & redraw everything        */

      removeClipRect( window,prevRegion );
      drawAll(window);

      /*  find new world extent and reset pan gadgets        */

      setWorldExtent( &world );
      setPanGadgets( window );
      return 0;
}
/*=========================================================
 *  dragAndRotate() - process rotate interaction
   =======================================================*/

int dragAndRotate( struct Window *window,
                   long mod,void *data )
{
   struct Region  *prevRegion;
   object_t *obj =    world.selected;
   double          rectxy[12],startAng;
   double          x1,y1,x2,y2,x,y,rcx,rcy;
   long            minx,miny,maxx,maxy;
   long            cx,cy,dx,dy;
   long            l,t,r,b;

   /*  If no object selected, just return     */

   if ( obj == NULL )
      return 0;

   /*
    *  clip to the view window rectangle, highlight the
    *  object, and draw rectangle around the object.
    */

   getViewExtent(window,&l,&t,&r,&b);
   prevRegion = setClipRect(window,l,t,r,b);

   SetAPen(window->RPort,HLPEN);
   point2dForward(world.viewTf,obj->minx,obj->miny,&x1,&y1);
   point2dForward(world.viewTf,obj->maxx,obj->maxy,&x2,&y2);
   minx = doubleToLong(x1);    miny = doubleToLong(y1);
   maxx = doubleToLong(x2);    maxy = doubleToLong(y2);
   drawObject(window,world.viewTf,obj);
   drawRect(window->RPort,minx,miny,maxx,maxy);

   /*
    *  set up the real valued rotation rectangle, find the
    *  rotation point, the start angle, and the delta
    *  angle, then call the drag handler.
    */

   rcx = (obj->minx+obj->maxx)/2.0;
   rcy = (obj->miny+obj->maxy)/2.0;
   rectxy[0] = rectxy[6] = obj->minx - rcx;
   rectxy[1] = rectxy[3] = obj->miny - rcy;
   rectxy[2] = rectxy[4] = obj->maxx - rcx;
   rectxy[5] = rectxy[7] = obj->maxy - rcy;
   rectxy[8] = rcx;  rectxy[9] = rcy;
   point2dForward(world.viewTf,rcx,rcy,&x,&y);
   cx = doubleToLong(x);    cy = doubleToLong(y);
   dx = world.pickX - cx;   dy = world.pickY - cy;
   rectxy[10] = startAng = atan2(-(double)dy,(double)dx);
   rectxy[11] = 0.0;

   handleDrag( window,(void*)rectxy,cx,cy,4,
               rotateDrag,&dx,&dy );

   /*
    *  erase object and rectangle, find delta rotation
    *  angle, and rotate the object.
    */

   SetAPen(window->RPort,BGPEN);
   drawObject(window,world.viewTf,obj);
   drawRect(window->RPort,minx,miny,maxx,maxy);
   rotateObject(obj,rectxy[10]-startAng);

   /*  remove the clip region & redraw everything        */
```

```
      removeClipRect( window,prevRegion );
      drawAll(window);

      /*  find new world extent and reset pan gadgets        */

      setWorldExtent( &world );
      setPanGadgets( window );
      return 0;
}
/*=========================================================
 *  zoomDrag() - drag/draw procedure for creating a
 *               zoom rectangle.
   =======================================================*/

void zoomDrag( struct Window *window,void *obj,
               long oldDx,long oldDy,
               long newDx,long newDy,long code )
{
   struct RastPort *rp = window->RPort;
   long *xy = (long*)obj;

   /*  draw previous rectangle                    */

   drawRect( rp,xy[0],xy[1],xy[2],xy[3] );

   if ( code == BEGINDRAG || code == ENDDRAG )
      return;

   /*  resize it and draw new rectangle            */

   xy[2] = xy[0] + newDx;  xy[3] = xy[1] + newDy;
   drawRect( rp,xy[0],xy[1],xy[2],xy[3] );
}
/*=========================================================
 *  moveDrag() - drag/draw procedure for dynamically
 *               moving a rectangle.
   =======================================================*/

void moveDrag( struct Window *window,void *obj,
               long oldDx,long oldDy,
               long newDx,long newDy,long code )
{
   struct RastPort *rp = window->RPort;
   long *xy = (long*)obj;
   long dx,dy;

   /*  draw at previous location                  */

   drawRect( rp,xy[0],xy[1],xy[2],xy[3] );

   if ( code == BEGINDRAG || code == ENDDRAG )
      return;

   /*  move it and draw at new location            */

   dx = newDx - oldDx;  dy = newDy - oldDy;
   xy[0] += dx;  xy[1] += dy;
   xy[2] += dx;  xy[3] += dy;
   drawRect( rp,xy[0],xy[1],xy[2],xy[3] );
}
/*=========================================================
 *  sizeDrag() - drag/draw procedure for dynamically
 *               scaling an object.
   =======================================================*/

void sizeDrag(struct Window *window,void *obj,
              long oldDx,long oldDy,
              long newDx,long newDy,long code )
{
   struct RastPort *rp = window->RPort;
   long *xy = (long*)obj;
   int wd2,hd2;

   /*  draw at previous location                  */

   drawRect(rp,xy[0],xy[1],xy[2],xy[3]);

   if ( code == BEGINDRAG || code == ENDDRAG )
      return;

   /*  resize the rectangle and draw it again      */

   wd2 = ABS(newDx);  hd2 = ABS(newDy);
   xy[0] = xy[4] - wd2;  xy[1] = xy[5] + hd2;
   xy[2] = xy[4] + wd2;  xy[3] = xy[5] - hd2;
   drawRect(rp,xy[0],xy[1],xy[2],xy[3]);
}
/*=========================================================
 *  rotateDrag() - drag/draw procedure for dynamically
 *                 rotating an object.
   =======================================================*/

void rotateDrag(struct Window *window,void *obj,
                long oldDx,long oldDy,
                long newDx,long newDy,long code )
{
   struct RastPort *rp = window->RPort;
   double *xy = (double*)obj;
   double newAng,deltaAng,cosa,sina,x,y,tmp;
```

```
        long px0,py0,px,py;
        int i;
        static double rad1 = degreesToRadians(1.0);

        /*
         * find new angle values.  If not first or last time,
         * and rotation is less than 1 degree, return.
         * otherwise, draw the rectangle
         */

        newAng = atan2(-(double)newDy,(double)newDx);
        deltaAng = newAng-xy[10];

        if ( code != BEGINDRAG && code != ENDDRAG &&
             ABS(deltaAng) < rad1 )
          return;

        point2dForward(world.viewTf,
                       xy[8]+xy[0],xy[9]+xy[1],&x,&y);
        px0 = doubleToLong(x);  py0 = doubleToLong(y);
        Move(rp,px0,py0);

        for( i = 2; i < 7; i+=2 )
        {
            point2dForward(world.viewTf,
                           xy[8]+xy[i],xy[9]+xy[i+1],&x,&y);
            px = doubleToLong(x);  py = doubleToLong(y);
            Draw(rp,px,py);
        }
        Draw(rp,px0,py0);

        if ( code == BEGINDRAG || code == ENDDRAG )
          return;

        /*
         * save angle values, rotate the rectangle around its
         * center (0,0), and redraw it.
         */

        xy[10] = newAng;  xy[11] = deltaAng;
        cosa = cos(xy[11]);  sina = sin(xy[11]);
        tmp = cosa*xy[0] - sina*xy[1];
        xy[1] = sina*xy[0] + cosa*xy[1];
        xy[0] = tmp;
        tmp = cosa*xy[2] - sina*xy[3];
        xy[3] = sina*xy[2] + cosa*xy[3];
        xy[2] = tmp;
        xy[4] = -xy[0];  xy[5] = -xy[1];
        xy[6] = -xy[2];  xy[7] = -xy[3];

        point2dForward(world.viewTf,
                       xy[8]+xy[0],xy[9]+xy[1],&x,&y);
        px0 = doubleToLong(x);  py0 = doubleToLong(y);
        Move(rp,px0,py0);

        for( i = 2; i < 7; i+=2 )
        {
            point2dForward(world.viewTf,
                           xy[8]+xy[i],xy[9]+xy[i+1],&x,&y);
            px = doubleToLong(x);  py = doubleToLong(y);
            Draw(rp,px,py);
        }
        Draw(rp,px0,py0);
}
/*==========================================================
 * moveObject() - move an object by adding delta
 *                coordinate values to the object.
 ==========================================================*/

void moveObject( object_t *object,double dx,double dy )
{
    double *poly = object->poly;
    int i;

    for ( i = 0; i < object->npts; i += 2 )
    {
        poly[i] += dx;  poly[i+1] += dy;
    }
    object->minx += dx;  object->miny += dy;
    object->maxx += dx;  object->maxy += dy;
}
/*==========================================================
 * resizeObject() - resize an object using a scaling
 *                  transform matrix.
 ==========================================================*/

void resizeObject( object_t *object,double sx,double sy )
{
    transform2_t tform;
    double *poly = object->poly;
    double tx,ty,minx,miny,maxx,maxy;
    int i;

    /*
     * set up the scaling matrix - translate the object
     * to the world origin, scale, then translate back
     * to original location.
     */
```

```
    tx = (object->minx+object->maxx)/2.0;
    ty = (object->miny+object->maxy)/2.0;
    mat2dInit( tform.matrix );
    mat2dTranslate( tform.matrix,-tx,-ty );
    mat2dScale( tform.matrix,sx,sy );
    mat2dTranslate( tform.matrix,tx,ty );

    minx = miny = 1.e38;
    maxx = maxy = -minx;

    for ( i = 0; i < object->npts; i += 2 )
    {
        point2dForward(&tform,poly[i],poly[i+1],
                       &poly[i],&poly[i+1]);
        minx = MIN(poly[i],minx); miny = MIN(poly[i+1],miny);
        maxx = MAX(poly[i],maxx); maxy = MAX(poly[i+1],maxy);
    }
    object->minx = minx;  object->miny = miny;
    object->maxx = maxx;  object->maxy = maxy;
}
/*==========================================================
 * rotateObject() -  rotate an object using a rotation
 *                   transform matrix.
 ==========================================================*/

void rotateObject( object_t *object,double ang )
{
    transform2_t tform;
    double *poly = object->poly;
    double minx,miny,maxx,maxy,tx,ty;
    int i;

    /*
     * set up the rotation matrix to rotate object around
     * its center - translate coordinates to center,
     * rotate, then undo the translation
     */

    tx = (object->minx+object->maxx)/2.0;
    ty = (object->miny+object->maxy)/2.0;
    mat2dInit( tform.matrix );
    mat2dTranslate( tform.matrix,-tx,-ty );
    mat2dRotate( tform.matrix,radiansToDegrees(ang));
    mat2dTranslate( tform.matrix,tx,ty );

    minx = miny = 1.e38;
    maxx = maxy = -minx;

    for ( i = 0; i < object->npts; i += 2 )
```


Pan a world view window?

```c
    {
        point2dForward(&tform,poly[i],poly[i+1],
                                &poly[i],&poly[i+1]);
        minx = MIN(poly[i],minx);  miny = MIN(poly[i+1],miny);
        maxx = MAX(poly[i],maxx);  maxy = MAX(poly[i+1],maxy);
    }
    object->minx = minx;  object->miny = miny;
    object->maxx = maxx;  object->maxy = maxy;
}
/*=========================================================
 *  setClipRect() - set a clipping rectangle in a window
 ========================================================*/

struct Region *setClipRect( struct Window *win,
                            long minx,long miny,
                            long maxx,long maxy )
{
    struct Region *newClip;
    struct Rectangle clip;

    newClip = (struct Region*)NewRegion();

    clip.MinX = minx;    clip.MinY = miny;
    clip.MaxX = maxx;    clip.MaxY = maxy;

    OrRectRegion(newClip,&clip);
    return ( (struct Region *)InstallClipRegion(
                            win->WLayer,newClip) );

}
/*=========================================================
 *  removeClipRect() - remove a previously installed
 *                     clipping rectangle from a window
 ========================================================*/

void removeClipRect(struct Window *win,struct Region *rgn )
{
    struct Region *oldClip;

    oldClip = (struct Region *)InstallClipRegion(
                            win->WLayer,rgn);
    if ( oldClip )
        DisposeRegion(oldClip);

}
/*=========================================================
 *  drawAll() - draw or redraw the world data, clipping
 *              to the window view rectangle
 ========================================================*/

void drawAll( struct Window *window )
{
    struct Region *prevRegion;
    object_t *obj;
    long l,t,r,b;

    /*  clip to the view window rectangle          */

    getViewExtent(window,&l,&t,&r,&b);
    prevRegion = setClipRect(window,l,t,r,b);
    drawAxis(window);
    SetAPen(window->RPort,DRAWPEN);

    for ( obj = world.objects; obj; obj = obj->next )
        drawObject(window,world.viewTf,obj);

    /*  remove the clip region                     */

    removeClipRect( window,prevRegion );

}
/*=========================================================
 *  eraseAll() - erase the world data by filling the
 *               window view rectangle with the
 *               background color.
 ========================================================*/

void eraseAll( struct Window *window )
{
    long l,t,r,b;
    long fg,bg;

    getViewExtent(window,&l,&t,&r,&b);
    fg = (long)window->RPort->FgPen;
    bg = (long)window->RPort->BgPen;

    SetAPen(window->RPort,0);  SetBPen(window->RPort,0);
    RectFill(window->RPort,l,t,r,b);
    SetAPen(window->RPort,fg);  SetBPen(window->RPort,bg);

}
/*=========================================================
 *  drawAxis() - draw the x,y world coordinate axes.
 ========================================================*/

void drawAxis(struct Window *window)
{
    double xp,yp;
    long l,t,r,b,cx,cy;

    /*  get the view window corner coordinates.       */

    getViewExtent(window,&l,&t,&r,&b);
```

```c
    SetAPen(window->RPort,AXISPEN);

    /*
     *  find world center in view coordinates and draw
     *  horizontal & vertical lines through the center
     *  coordinates.
     */

    point2dForward( world.viewTf,0.0,0.0,&xp,&yp);
    cx = doubleToLong(xp);  cy = doubleToLong(yp);
    if ( t < cy && cy < b )
    {
        Move(window->RPort,l,cy);
        Draw(window->RPort,r,cy);
    }
    if ( l < cx && cx < r )
    {
        Move(window->RPort,cx,t);
        Draw(window->RPort,cx,b);
    }
}
/*=========================================================
 *  findObject() - search for an object at the given
 *                 world coordinates.
 ========================================================*/

object_t *findObject( object_t *objList,double x,double y,
                      double eps )
{
    object_t *obj;

    for ( obj = objList; obj; obj = obj->next )
        if ( point2Poly( obj,x,y ) <= eps )
            return obj;

    return NULL;

}
/*=========================================================
 *  point2Poly() - find smallest distance from a point to
 *                 a polygon
 ========================================================*/

double point2Poly( object_t *obj,double x,double y )
{
    double *poly = obj->poly;
    double x1,y1,x2,y2;         /*  line segment endpoints */
    double distSq,minDistSq;    /*  squared distances      */
    double xStar,yStar;         /*  nearest points on seg  */
    double t1,deltaX,deltaY;
    double num,denom;
    int i;

    x1 = poly[0];  y1 = poly[1];
    if ( x == x1 && y == y1 )
        return 0.0;

    minDistSq = 1.e38;

    /*  find distance from point to each line segment     */

    for ( i = 2; i < obj->npts; i += 2 )
    {
        x2 = poly[i];  y2 = poly[i+1];
        if ( x == x2 && y == y2 )
            return 0.0;

        deltaX = x2 - x1;  deltaY = y2 - y1;
        denom = deltaX*deltaX + deltaY*deltaY;

        if ( denom < 1.e-6 )  /*  segment is a point      */
            continue;

        num = deltaX*(x-x1) + deltaY*(y-y1);
        t1 = num / denom;

        if ( t1 < 0.0 )    /*  x,y is left of segment     */
        {
            xStar = x1;  yStar = y2;
        }
        else
        if ( t1 < 1.0 )        /*  x,y is on the segment   */
        {
            xStar = x1 + t1*deltaX;  yStar = y1 + t1*deltaY;
        }
        else                   /*  x,y is right of the segment*/
        {
            xStar = x2;  yStar = y2;
        }
        distSq = (xStar-x)*(xStar-x) + (yStar-y)*(yStar-y);

        if ( distSq < minDistSq )
            minDistSq = distSq;

        x1 = x2;  y1 = y2;
    }
    return sqrt( minDistSq );

}
```

```c
/*=========================================================
 *   drawObject() - transform world coordinates to view
 *                   coordinates and draw a polygon.
 =========================================================*/

void drawObject(struct Window *window,transform2_t *tForm,
                object_t *object)
{
    double *poly = object->poly;
    double tx,ty;
    long px,py;
    int i;

    point2dForward( tForm,poly[0],poly[1],&tx,&ty );
    px = doubleToLong(tx);    py = doubleToLong(ty);
    Move( window->RPort,px,py );

    for ( i = 2; i < object->npts; i += 2 )
    {
        point2dForward( tForm,poly[i],poly[i+1],&tx,&ty );
        px = doubleToLong(tx);  py = doubleToLong(ty);
        Draw( window->RPort,px,py );
    }
}
/*=========================================================
 *   drawRect() - draw a rectangle using current pen & mode.
 =========================================================*/

void drawRect( struct RastPort *rp,
               long x1,long y1,long x2,long y2 )
{
    Move( rp,x1,y1 );
    Draw( rp,x1,y2 );    Draw( rp,x2,y2 );
    Draw( rp,x2,y1 );    Draw( rp,x1,y1 );
}
/*=========================================================
 *   setWorldExtent() - find and set world min/max values
 =========================================================*/

void setWorldExtent( world_t *world )
{
    object_t *obj;

    world->minx = world->miny = 1.e38;
    world->maxx = world->maxy = -1.e38;

    for( obj = world->objects; obj; obj = obj->next )
    {
        world->minx = MIN(world->minx,obj->minx);
        world->miny = MIN(world->miny,obj->miny);
        world->maxx = MAX(world->maxx,obj->maxx);
        world->maxy = MAX(world->maxy,obj->maxy);
    }
}
/*=========================================================
 *   getViewExtent() - return view window min/max values.
 =========================================================*/

void getViewExtent(struct Window *win,
                   long *l,long *t,long *r,long *b )
{
    *l = (long)win->BorderLeft + 1;
    *r = (long)(win->Width - win->BorderRight) - 1;
    *t = (long)win->BorderTop + 1;
    *b = (long)(win->Height - win->BorderBottom) - 1;
}
/*=========================================================
 *   handleInput() - monitor user input until end action
 =========================================================*/

void handleInput( struct Window *window )
{
    intuiExtension_t    *ext;    /* extension pointer */
    myMenuItem_t        *mi;
    struct IntuiMessage *msg,mcopy;
    struct Gadget       *gadg;

    for(;;)
    {
        Wait( 1 << window->UserPort->mp_SigBit );

        while( msg = (struct IntuiMessage *)
                            GetMsg(window->UserPort) )
        {
            mcopy = *msg;
            ReplyMsg( msg );

            /*
             *  Get our extension data from the Intuition
             *  object, then call the event handler if
             *  there is one.
             */

            ext = NULL;

            if ( mcopy.Class == MENUPICK &&
                 mcopy.Code != MENUNULL )
            {
                mi = (myMenuItem_t *)ItemAddress
```

```c
                            (window->MenuStrip,(LONG)mcopy.Code);

                ext = &mi->extens;
            }
            else
            if ( mcopy.Class == GADGETDOWN ||
                 mcopy.Class == GADGETUP )
            {
                if ( gadg = (struct Gadget*)mcopy.IAddress )
                    ext = (intuiExtension_t*)gadg->UserData;
            }
            else
                ext = (intuiExtension_t*)window->UserData;

            if ( ext && ext->handler )
                if ( (*ext->handler)(&mcopy,ext->data) )
                    goto allDone;
        }
    }

    /*  empty the message queue                         */

allDone:

    while( msg = (struct IntuiMessage *)
                          GetMsg(window->UserPort) )
        ReplyMsg( msg );
}
/*=========================================================
 *   handleDrag() - process input while user 'drags' an
 *                  object.
 *   NOTE:  NewWindow must have the REPORTMOUSE flag set
 *          (Flags) for this procedure to work correctly.
 =========================================================*/

void handleDrag( struct Window *win,void *object,
                 long startx,long starty,long moveEps,
                 void (*dragDraw)(),long *dx,long *dy )
{
    struct RastPort     *rp = win->RPort;
    struct IntuiMessage *msg;
    unsigned long       event,oldFlags,newFlags;
    unsigned short      code;
    long                oldMode,oldPen,oldDx,oldDy;
    long                testx,testy,oldx,oldy;
    long                newx,newy;

    /*
     *  save current IDCMP flags and set up for MOUSEBUTTONS,
     *  MOUSEMOVE, and MENUVERIFY events.
     */

    if ( moveEps <= 0 )
        moveEps = 1;

    oldx = startx;  oldy = starty;
    oldDx = oldDy = 0;

    oldFlags = win->IDCMPFlags;
    newFlags = MOUSEBUTTONS|MOUSEMOVE;
    if ( win->MenuStrip )
        newFlags |= MENUVERIFY;

    ModifyIDCMP( win,newFlags );

    /*
     *  save rastport pen & drawmode, then set pen to
     *  background pen and mode to COMPLEMENT
     */

    oldMode = (long)rp->DrawMode; oldPen = (long)rp->FgPen;
    SetDrMd(rp,COMPLEMENT);  SetAPen(rp,BGPEN);

    /*
     *  get drag started with initial call and process
     *  msgs until SELECTDOWN event occurs.
     */

    (*dragDraw)(win,object,0,0,0,BEGINDRAG);

    while( TRUE )
    {
        msg = (struct IntuiMessage *)GetMsg(win->UserPort);
        if ( ! msg )
            continue;

        /*  get values we need and reply to the message   */

        newx = (long)msg->MouseX;  newy = (long)msg->MouseY;
        testx = ABS(newx - oldx);  testy = ABS(newy - oldy);
        *dx = newx - startx;       *dy = newy - starty;
        event = msg->Class;        code = msg->Code;

        /*
         *  if we got a menu verify event, tell Intuition
         *  to cancel it.
         */

        if ( event == MENUVERIFY )
            msg->Code = MENUCANCEL;
```

```c
    ReplyMsg( (struct Message*)msg );

    /*
     * if cursor moved by specified amount, call the
     * dragdraw procedure, then save delta values
     */

    if ( testx >= moveEps || testy >= moveEps )
    {
        (*dragDraw)(win,object,oldDx,oldDy,*dx,*dy,0);
        oldx = newx;   oldy = newy;
        oldDx = *dx;   oldDy = *dy;
    }
    if ( code == SELECTDOWN )
        break;
}

/*  call dragDraw to finalize the drag            */

if ( oldDx != *dx || oldDy != *dy )
    (*dragDraw)(win,object,oldDx,oldDy,*dx,*dy,0);

(*dragDraw)(win,object,*dx,*dy,*dx,*dy,ENDDRAG);

/*  restore everything                            */

SetDrMd(rp,oldMode);    SetAPen(rp,oldPen);
ModifyIDCMP(win,oldFlags);
}
/*==========================================================
 *  openLibs() - open libraries
==========================================================*/

int openLibs( void )
{
    IntuitionBase = (struct IntuitionBase *)OpenLibrary(
                        "intuition.library",INTUITION_REV);
```

```c
    GfxBase = (struct GfxBase *)OpenLibrary(
                        "graphics.library",GRAPHICS_REV);
    LayersBase = (struct LayersBase *)OpenLibrary(
                        "layers.library",0L);
    return( GfxBase && IntuitionBase && LayersBase );
}
/*==========================================================
 *  closeLibs() - close libraries
==========================================================*/

void closeLibs( void )
{
    if ( IntuitionBase ) CloseLibrary(IntuitionBase);
    if ( GfxBase ) CloseLibrary(GfxBase);
    if ( LayersBase ) CloseLibrary(LayersBase);
}
/*==========================================================
 *  displayWindow() - display an intuition window
==========================================================*/

struct Window *displayWindow(int l,int t,int w,int h,
                             char *name,struct Gadget *gad)
{
    struct NewWindow  new;

    new.LeftEdge = l;   new.TopEdge = t;
    new.Width = w;      new.Height = h;
    new.DetailPen = -1; new.BlockPen = -1;
    new.Title = name;
    new.Flags = WINDOWCLOSE|WINDOWDRAG|SMART_REFRESH|
                ACTIVATE|WINDOWSIZING|REPORTMOUSE;
    new.IDCMPFlags = CLOSEWINDOW|REFRESHWINDOW|NEWSIZE|
                MENUPICK|MOUSEBUTTONS|GADGETUP;
    new.FirstGadget = gad; new.CheckMark = NULL;
    new.MinWidth = 150;    new.MinHeight = 100;
    new.MaxWidth = 640;    new.MaxHeight = 400;
    new.Type = WBENCHSCREEN;
    return( (struct Window *)OpenWindow(&new) );
}
```

C MACROS
for ARexx?

by David Blackwell

ARexx

Since its release, ARexx has become increasingly popular and the number of commercial programs supporting ARexx continues to grow. In fact, Commodore has licensed ARexx for inclusion with the AmigaDOS 2.0 operating system. ARexx is here to stay, and it will have a continued impact on all types of software developed for the Amiga. This can be attributed to the way ARexx macros are used to extend the applicability of an existing program or to combine several programs into a user-customized environment.

ARexx macros can be called by a host program to perform a specific task the program is either unable to perform itself or performs inefficiently. This is the way an ARexx macro can extend a program's utility. This capability of ARexx makes it possible for programmers to highly specialize the host program's operation. Programmers can optimize their code for speed of execution and reduce its size. Users can then add the frills or features they want or need through the addition of ARexx macros. All the programmer needs to do is include the capability in the host program to call ARexx macros and process their responses: in other words, an ARexx interface. An ARexx interface is relatively simple to include in the host program and usually increases the code size by about 2-3K (most likely less than the size saved by optimizing the program's operations).

The most advanced ARexx macros are written to combine several application programs into a totally customized environment. One macro—or perhaps a group of macros— directs the efforts of several programs in any way the user can imagine. This capability of ARexx macros staggers the imagination and has yet to be fully realized. As the number of programs supporting ARexx increases, and as users become better acquainted with ARexx, these macros will become more commonplace. The release of AmigaDOS 2.0 should greatly accelerate this eventuality.

ARexx and C

Although ARexx is powerful and easy-to-learn, some of us don't have the time or the desire to learn a new programming language. We are quite happy with the programming language we currently use. Fortunately, if your favorite language is C, you can still have access to the power ARexx provides (Modula 2 and assembly language interfaces are also possible). As a matter of fact, the ARexx interpreter is a shared library (also known as a resident or system library). This library allows access to all the essential ARexx functions. There are two ways to access these functions using C: glue routines and pragmas.

Glue Routines

By using the glue routines provided on the ARexx commercial disk, it is a simple matter to call these functions. When you call any ARexx system library function, you are actually calling a small assembly language routine. This routine is called a glue routine. The glue routine pulls the arguments off the stack, puts the arguments into the appropriate registers, puts the library base pointer into the a6 address register, and then calls the library routine with the proper library vector offset. As you can imagine, these glue routines add a little overhead to your program in terms of size and speed. If you have an ANSI C compiler, you have another option: use pragmas.

Pragmas

I prefer to use pragmas since they neither increase the size of the program, nor slow it down. The ANSI standard allows for extensions to the C language in a controlled manner using pragmas. By using a pragma, the C compiler will generate code to call the library function directly without pushing any arguments onto the stack or using a glue routine. All arguments are loaded directly into the proper registers and the →

library's base address is placed in the a6 address register. Then the function is called using the proper library vector offset (Aztec C-style pragmas for the ARexx functions I used in my example programs are included in the Graphics.h file).

Whether you use glue routines or pragmas, you still have the same access to the ARexx functions using C. The only difference will be in code size and speed of execution. There are, however, disadvantages and advantages to using C rather than ARexx itself.

Disadvantages

Programming a macro in C will require more code than using ARexx:

```
address 'hostname'
```

in ARexx, for example, does the same thing as,

```
Forbid();
if ( (hostport = FindPort(HOSTPORTNAME)) == 0 )
{
    Permit();
    return( <error number> );
}
Permit();
...
```

Also, the ARexx interpreter automatically allocates any message ports your macro requires. This operation is totally transparent to the ARexx macro programmer. On the other hand, in C you are required to set up your own message ports, keep track of any message port activity and dispose of the message ports when you no longer need them. This difference alone can generate a lot of code. At a minimum, you will need four functions to manage your message ports: one function to open the message port, one function to get messages from the message port, one function to send messages to another port and one function to release the message port when you're done with it. There is a way to turn this particular disadvantage into an advantage; however, we'll discuss that later.

Perhaps one of the biggest disadvantages of C is the lack of a built-in tracing facility. With ARexx you have eight distinct tracing options, and you can even selectively debug a section of your program while all other sections execute without tracing. Perhaps the most missed feature is the interactive tracing offered by the ARexx interpreter.

With these apparent disadvantages one may question the use of C in the first place. Well, in my opinion, the advantages of using C far outweigh the disadvantages.

Advantages

Perhaps the most noticeable advantage is the execution speed of a macro written in C. For small macros this might not be too apparent, but when you start writing sizeable, mature macros you will begin to notice a marked difference in their performance. To be fair, no interpreted language can match the speed of a compiled language. When you start expecting a lot from your macros, you will really appreciate the increase in speed.

Also, the more you begin to expect from a macro, the more you will want the power and control that C affords you. Not only do you have all the access to the high-level Intuition, Exec and AmigaDOS routines that ARexx provides but, if you need it, you have access to the low-level Graphics and AmigaDOS routines as well. You decide just how much control over the Amiga you can handle. Sometimes, too much power in the wrong hands is a dangerous thing. However, an experienced programmer can really take control of the Amiga and do some amazing things with an ARexx macro written in C. Besides speed and power, code integrity is a real concern for some programmers.

If you are really proud of what you have accomplished with your macro, and program code integrity is a high priority of yours, you will really appreciate the fact that C macros are extremely difficult to tamper with. In fact, the only way your work can be tampered with is if you release the source code with the executable program. After you have spent days, weeks or perhaps even months working on a macro and getting it just as you want it, the last thing you want to do is release it into the public domain and instantly have in excess of 100 different versions of it out there, with a multitude of various users requesting help from you or blaming you for a problem you didn't even know existed. With code integrity, all the improvements and additions to your macro must go through you. And that's how it should be for any program that the original programmer intends to fully support. But enough with this discussion on the disadvantages and advantages of C. Let's move on to actual macro development in C.

Macro Development

Now for the hands-on stuff. Let's get started actually examining a macro written in C. This macro (actually two macros were prepared for you) will be designed to work with a program that has a message port and can process ARexx messages. If you received the Premiere issue of *AC's TECH*, refer to the article by Dan Sugalski concerning interprocess communication with ARexx. I have converted Mr. Sugalski's ARexx macros to C as a comparison of the two languages. If you didn't get the Premiere issue but have the disk for the second issue, these programs can be found on that disk in the archive directory under the name "IPC.lzh". You will want to look at the Example3.rexx and Example4.rexx programs. (If you don't have either the Premiere issue or the disk to the second issue, and are extremely interested in seeing the difference between the two languages I'm sure there are back issues available.)

One of the first things you need to do in a C macro that will access the ARexx system library is to declare a Global variable to hold a pointer to the base address to that library.

```
struct RxsLib *RexxSysBase;
```

That should do very nicely. The name must be exactly as shown. When you make the call to the OpenLibrary() routine, you must cast the return value to the correct structure or your compiler will complain.

```
RexxSysBase = (struct RxsLib *) OpenLibrary(RXSNAME, 0L);
```

The above example is the proper way to accomplish that task. This library must be opened before you attempt to call any of the ARexx functions, unless you enjoy visits from the Guru.

I call a function to open all the libraries I will use in my program similar to the function described by Paul Castonguay in his article, "Programming the Amiga's GUI in C, Part I," in the second issue of *AC's TECH*. This is a good structured approach to programming reusable functions which only require slight modifications to be used in any program you write. This approach will be very important in writing C macros since many of the routines required to get set up and communicate with another program can be written once and used repeatedly to save programming time. In fact, I used that very approach on the two macros I converted as you will see when you examine the code. After opening the ARexx system, you are ready to open a message port.

Now a major design decision must be made. Do you want to allow multiple copies of your macro to run at the same time, or do you just want one copy running at a time? How you open your message port will determine this. For my example program I chose single execution. In that design methodology, you first check for the existence of a message port by the name you want yours to be. If one exists, you return an error condition and terminate execution of the macro. Under the multiple-copy methodology, you would simply add a unique suffix to each new message port name so each could be easily distinguished.

Because of the importance of a message port to the successful operation of an ARexx macro, your routine that opens your message port must inform the main program whether it was successful. If it was successful, then execution continues as normal. If it was unsuccessful, then the macro should exit gracefully. Also, here is where using C can be an advantage to ARexx opening a message port for you. With ARexx, you can only communicate through one message port at a time. Using C, you can communicate through as many message ports as you like all at once, and you will be able to pull it all off much quicker than ARexx.

Opening the ARexx system library and opening a message port are the most basic setup requirements. Your particular macro may require a more elaborate setup; however, that is up to each individual programmer. After setup, though, macro operations must commence.

Macro Communications
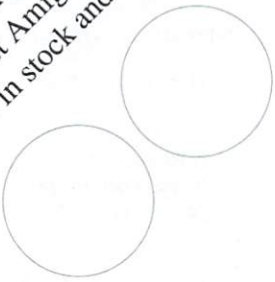
Macro operations consist of communications between your macro program and the host program with which it was designed to operate. Here another major design decision must be made. This time it is a choice between either synchronous or asynchronous communications. Simply put, it is a choice between sending a message and then waiting for a response before doing anything else (synchronous), or sending a message and continuing with other processing, occasionally checking your message port for a response or waiting on a signal from your message port after all other processing you

can do is completed. Also, you need a response before you can continue (asynchronous). I chose asynchronous communications for my example programs. I just don't like waiting around for a response when there are other things I could be doing. Here is where you have another advantage over a macro written in ARexx. ARexx uses synchronous communications whether you want it or not; C gives you a choice. Once the decision is made concerning which type of communications you will use you can start designing your message-send function. Again, a fairly standard function can be designed here so that, with a little modification, it can be used in any program you write in the future.

Here are the design considerations I used when developing my message-send function:

1. The function should receive as few arguments as possible.
2. The existence of the receive port should be checked.
3. In case of error, all memory allocated should be released.
4 If all goes well, send message to receive port.

And here is the code I used to satisfy my design considerations:

```
void arexxsend(STRPTR commandstr)
{
        struct MsgPort *sendport;
        struct RexxMsg *rexxmsg;

        if (!(rexxmsg = CreateRexxMsg((struct RexxMsgPort *)Graph1,\
                    (STRPTR) 0, (STRPTR) MYPORT1)))
            goto rs_exit; /* not able to create a message */

        if (!(rexxmsg->rm_Args[0] = CreateArgstring(commandstr,\
                    (ULONG) strlen((char *) commandstr))))
        { /* not able to create argstring so delete */
            DeleteRexxMsg(rexxmsg); /* the RexxMsg and */
            goto rs_exit; /* leave the routine */
        }

        rexxmsg->rm_Action = RXCOMM; /* indicate a command */

        Forbid();

        if (!(sendport = FindPort(MYPORT1)))
        {
            Permit(); /* unable to find the port so delete the */
            DeleteArgstring(rexxmsg->rm_Args[0]); /* Argstring */
            DeleteRexxMsg(rexxmsg); /* and the RexxMsg */
            goto rs_exit; /* and leave the routine */
        }

        /* everything went ok so send the message */
        PutMsg(sendport, (struct Message *) rexxmsg);

        Permit();

rs_exit:

        return;
}
```

The CreateRexxMsg() function does just what it implies: it creates an ARexx message structure to communicate with other ARexx-compatible programs. The first argument is a pointer to the message port structure to use as a reply port for the message, and the last two arguments are pointers to null-terminated strings. The first string is the default file extension to be used when requesting ARexx to load and execute a macro, and the second string the initial host address. This function allocates all the memory necessary to create the message structure. The CreateArgstring() function creates an ARexx argument string using the supplied string. The first argument is a pointer to the null-terminated string to use, and the second argument is the length of the string. This function also allocates all the memory that it needs to create the argument string structure.

You may have noticed that I disable task-switching and check for the existence of the send port before I attempt to actually send the message. This is very important on a multitasking computer where programs can terminate as quickly as they began. By disabling task switching and ensuring that the program is actually there before sending the message, you avoid a lot of problems and even a possible Guru visit or two. If the receive port has closed, I release the memory I just allocated by using the DeleteRexxMsg() and DeleteArgstring() functions which use the pointers returned by the earlier functions and return to the calling program.

Finally, if all goes well the message is sent on its way to receive prompt attention. The last thing I do before returning is enable task-switching again.

A small task for you: if I had decided on synchronous communications, how would I have to modify this function to achieve that? You could even have two send functions—one synchronous and one asynchronous—in the same macro, and use the one best suited for each particular command. Also, how would you modify this function so that it returns an error status to indicate whether it was successful or not?

In my small examples I don't really care about any return values; in fact, I don't even check for any, much less set any return values. On the other hand, in an important macro you would always check the return values. You would check the rm_Result1 field of the RexxMsg structure to see if your command or request was successfully executed. If you requested a return string from the host, you would copy it to a safe place using the string pointer in the rm_Args[0] field. After you copy the string to a safe place, you can release the argument string and message structure memory. You would then take appropriate action depending on the value you extracted from the rm_Result1 field. These are the absolute basics to macro operations. No matter how elaborate your macros become, these are the basic building blocks of all macros. When your macro is finished with its task, what do you do then?

Macro Shutdown

This should answer your question—you close up shop and terminate operations. That is as easy as deleting all memory allocated during macro operations, closing down your message port, closing any windows opened and then closing the ARexx system library. It is always a good idea when deleting memory and closing your message port to disable task-switching and avoid the possibility of other programs trying to send you messages while you are attempting to close down. Just don't forget to enable task-switching again before you terminate execution of your macro. My Release_Port() function demonstrates one way to do this:

```
void Release_Port(void)
{
        struct RexxMsg *rtnmsg;

        Forbid();

        /* delete all outstanding messages */
        while (rtnmsg = (struct RexxMsg *) GetMsg(Graph1))
        {
            if (rtnmsg->rm_Node.mn_Node.ln_Type == NT_REPLYMSG)
            { /* a reply to one of my messages so delete it */
                DeleteArgstring(rtnmsg->rm_Args[0]);
                DeleteRexxMsg(rtnmsg);
            }else{ /* not one of my messages so reply to it */
```
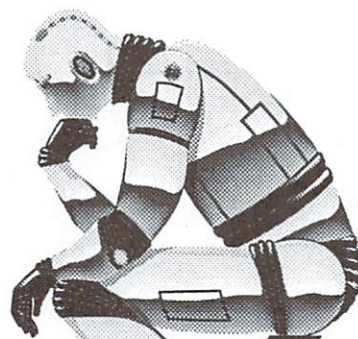
→

```
        rtnmsg->rm_Result1 = RC_ERROR;  /* indicate error */
        ReplyMsg((struct Message *)rtnmsg);

   }

 }

 DeletePort(Graph1);  /* delete the port */

 Permit();
 return;

}
```

Running the Macro

ARexx macros are normally executed from the CLI by typing the command "rx" followed by the program name and any arguments. C macros do not need to be preceded by the "rx" command. They can be executed directly from the CLI prompt. This raises a small problem if you are writing the macro to be executed by a host program directly rather than executing the macro from the CLI prompt. Host programs run macros by sending a message to the ARexx-resident process requesting that the macro be executed. All ARexx macros are basically ASCII files that are interpreted and executed by the ARexx interpreter. On the other hand the C macro, once compiled, is an executable program. The ARexx interpreter cannot execute a compiled C macro. If you are writing your macro to be executed by a host program, you will need a small ARexx boot program to start your compiled C macro. This is not much trouble, and you can use the same small boot program to start other C macros you write. The following code is all you need to have an ARexx boot program:

```
/* ARexx boot program for the compiled Graphics1 C macro */
address command "Graphics1"
exit
```

That's all it takes to have your C macro executed by ARexx. To use the same code with other C macros, simply change the program name in quotation marks to the name of the program you want executed. If your macro requires arguments, include them in the quotation marks. Perhaps if the use of compiled languages to write ARexx macros increases, commercial programmers will start including the option to execute ARexx macros directly without using the ARexx interpreter.

Conclusion

As stated at the onset of this article, ARexx's popularity continues to grow. By becoming familiar with it, you can more easily customize those commercial programs you purchase which have ARexx interfaces. You will get a better end product by adapting the feel of the program to your personality.

You can use the ARexx interpreted language to write your macros, or you can use C when you need the advantages of speed, power, and code integrity. The capability to open several message ports and choose asynchronous communications, if needed, cannot be overlooked.

Although using C requires you to write more code, the positive side to that is if you use structured programming techniques, you can reuse the functions repeatedly with just slight modifications. This will save you time in the long run

and results in making programming in C almost as easy as programming in ARexx. Also, if you are already familiar with C, you save the time of learning a new language.

Other Notes

These example programs were not meant to show you all the ins and outs of C macro programming for ARexx, but to get you started and then let you learn the finer points yourself. These programs have no value other than as instructional examples.

I realize that some may be confused with my encouragement of structured programming while at the same time I use the abhorred goto statement throughout my programs. This is simple to explain. I believe highly in structured programming, but I also believe that as intelligent beings, we should not be stifled by stringent inanimate principles. I also disdain the undisciplined use of the goto statement, and I believe that I use it in a very structured manner. I attempt to use a programming style that is easy to use and understand. I will show some alternatives to my use of the goto's as a comparison and contrast and let you decide which you think is better:

Alternative 1 - The If-Then-Else ad infinitum

```
if (no error)
   continue processing
   if (no error)
      continue processing
      if (no error)
         continue processing
                ad infinitum
      else
            set error condition
   else
         set error condition
else
      set error condition
```

These things can soon indent themselves right off the screen and create a real mess when you try to print them.

Alternative 2 - The multiple exit points from a routine

```
if (error)
   display error
   exit program/function
endif

if (error)
   display error
   exit program/function
endif

continue processing
```

This is similar to my use of the goto statement but, in my opinion, it is a violation of the basic structured programming premise that each program or routine should have only one entry point and one exit point. However, each programmer eventually decides for himself or herself which style they feel more comfortable with.

☑

VBRMon

An Interesting and Unusal Assembly Language Monitor

by Dan Babcock

VBRMon is an exciting utility that will help you more fully explore and take control of your Amiga. It is a machine language monitor like many others, but it has some very interesting and unusual properties...

The Concept

VBRMon takes advantage of the vector base register (VBR) present in the 68010 and higher 68000-family CPU's. The vector base register, quite simply, allows one to move the exception vector table (always the lowest 1K of memory on a plain 68000) anywhere in the 32-bit address space. (See Figure One for a map of the exception table.) Why is this useful? Consider that all (or almost all) software assumes that the exception table is always located at location 0. The relocated vector table, therefore, is essentially protected and we can intercept all exception activity (most important, interrupts). That means that whenever an interrupt occurs (which is quite often on the Amiga!) our devious little exception handler can grab control. In practice, this exception handler will check to see if both mouse buttons are pressed. If so, it will invoke a machine language monitor to enable the examination of the currently running program(s). If not, it simply jumps to the corresponding "real" vector, located in low memory. Thus, it is almost totally transparent to the host software.

Typical Exception "Wedge"

To illustrate the concept, the following is a typical exception "wedge" routine, which will be executed whenever an exception (or an interrupt, which is a class of exception) occurs. It is a simplified version of the one in VBRMon. Note

that this routine is universal—it can replace any or all of the exception vectors (except bus error). A special routine for each vector is not necessary. Figure Two shows the stack arrangement upon entering this routine.

```
Handler:

    btst    #6,($bfe001)        ;check left mouse button
    beq.b   Hit                 ;go if pressed
    move.l  a0,-(sp)            ;any used registers must be saved
    move.w  (10,a7),a0          ;load vector offset into a0
    move.l  (a0),-(sp)          ;fetch "real" vector, and set
                                ;up for RTS
    move.l  (4,a7),a0           ;restore a0
    rtd     #4                  ;This is a special RTS that cleans
                                ;up the stack (adds 4 to SP).
                                ;[Only available on '010 or higher]
Hit:
;Force the background color to red to let the programmer know his code works!
    move.w  #$0f00,($dff180)
    rte
```

That's all there is to it!

Requirements

First of all, VBRMon works best if you have a 68010—no surprise there! However, it does provide some utility on plain 68000 machines. It will still trap gurus and the Debug() vector, and it permits you to use a "GOMF button" (i.e., an NMI-generating device) to invoke VBRMon. Second, it works best with 1MB of Chip RAM available. Normally the program code, its data, and the screen display are kept in the upper 512K of Chip RAM, leaving all the lower 512K available for the use of the host application (e.g., game), and also facilitating reboot survival.

➤

How To Start It

VBRMon may be started by simply typing "vbrmon" at the CLI prompt or by clicking on its icon—yes, it supports Workbench! You will then be asked to enter addresses for the program code, data area, and screen memory. If you have 1MB of chip memory, the defaults should prove acceptable. It is also possible to specify the desired options as command line parameters, like so:

```
VBRMon Program_address Screen_address Data_address
```

The addresses should be entered in hex, without a preceding $. If all goes well, you will be congratulated with the message, "VBRMon initialized successfully. Press both mouse buttons to invoke." If you have only a plain 68000, you will receive a somewhat less satisfying message.

Using It

Assuming that you have a 68010 or higher, you may now invoke VBRMon by pressing both mouse buttons at any time. The vector that was trapped will be displayed (e.g., "VBRMon invoked by vector $6C") along with a register dump, including the current program counter, and INTENAR and DMACONR. You are now in control of a powerful machine language monitor with which you may examine and alter the state of the machine. Use the control key to pause screen output. Delete places the cursor at the top of the screen; shift-delete clears the screen. Escape will abort most commands. Use shift-cursor-up and shift-cursor-down to access previously entered lines. Note that VBRMon is fully independent of the operating system. Try it on your favorite take-over-the-system game!

Figure One (Source: *Programming The 68000* by Steve Williams)

Vector	Address	Function
0	$0	RESET initial SSP (supervisor stack pointer)
1	$4	RESET initial PC (program counter)
2	$8	Bus error
3	$C	Address error
4	$10	Illegal instruction
5	$14	Divide by zero
6	$18	CHK instruction
7	$1C	TRAPV instruction
8	$20	Privilege violation
9	$24	Trace
10	$28	Line 1010 emulator
11	$2C	Line 1111 emulator
12-14	$30-$38	Unassigned (reserved)
15	$3C	Uninitialized interrupt vector
16-23	$40-$5C	Unassigned (reserved)
24	$60	Spurious interrupt
25-31	$64-$7C	Level 0-7 autovector interrupts
32-47	$80-$BF	Trap 0-15 vectors
48-63	$C0-$FC	Unassigned (reserved)
64-255	$100-$400	User interrupt vectors (not used)

Command Rundown

Press the HELP key for a command overview while in VBRMon. Here's a brief explanation of the commands:

```
Function: Assemble
Syntax:   A <starting address>
```

This is a mini-assembler for constructing small (but useful) routines and code patching on-the-fly. It accepts standard 68000 instructions in the usual (old) syntax.

```
Function: Change or view numeric base
Syntax:   B <byte>
```

This function changes the default numeric base used in expressions. (All commands accept expressions in addition to simple numbers; see the ? command description for a list of operators.) The input is always in decimal, which allows you to

Figure Two

Stack Arrangement Upon Entering Exception Wedge Routine

A7	->	status register before exception (WORD)
		program counter (LONG)
		vector offset (WORD)

back out of the otherwise catastrophic state of being in base 3 or another useless numeric base. Base 16 is hexadecimal (the default), base 10 is decimal, and base 2 is binary.

```
Function:  Compare memory, listing differences
Syntax:    C start1 end1 start2
```

The compare command compares the corresponding bytes in two blocks of memory, listing the address of each different byte in the first block. Note that the addresses are *not* stored in the hunt buffer (see the L command).

```
Function:  Disassemble 68000 instructions
Syntax:    D <start> <end>
```

The disassemble command disassembles standard 68000 instructions into their mnemonic equivalent, using the usual (old) syntax.

```
Function:  Fix ExecBase and reinstall VBRMon
Syntax:    E
```

This command is typically used after executing some program (e.g., a take-over-the-system game) that smashes ExecBase. It will restore the original ExecBase (saved when VBRMon was first invoked on the command line) and also reinstall VBRMon into the KickTagPtr/KickMemPtr fields. This can be an extremely convenient feature!

```
Function:  Fill memory with pattern
Syntax:    F start end byte <byte> <byte> ...
```

The memory range indicated by the start and end addresses will be filled with the byte string supplied. Use single quotes to indicate a text string.

```
Function:  Joystick autofire
Syntax:    FIRE <port>
```

The feature is provided for your gaming amusement. It provides a steady stream of "fire" by toggling the joystick/mouse fire bit in the CIA, which this function configures as an output. It is okay to press the button while this function is active; no hardware damage will occur.

The <port> parameter is the port number, either 0 or 1. Invoking FIRE with no parameters turns all autofire off.

```
Function:  Enable guru trapping
Syntax:    GURU
```

This function patches vectors 3, 4, 5, 10, and 11 to immediately invoke VBRMon without checking the mouse buttons. This is extremely valuable for debugging purposes. Guru trapping may be disabled by a second invocation of this function.

Note that a different kind of guru trapping is always available. If you reboot (control-amiga-amiga) after receiving a guru meditation alert, VBRMon will automatically be invoked, and the register set (stored by Exec at $180) will be available for viewing. This is a more passive approach than the GURU function.

```
Function:  Hunt
Syntax:    H start end <byte> <byte> ...
```

The hunt command searches a memory range for any and all occurrences of the specified byte string, printing the addresses of any matches. In addition, the first few match addresses are stored in the hunt buffer, which may be viewed using the L command. The syntax is the same as for the fill command.

```
Function:  Info about this program
Syntax:    INFO
```

Displays a few programming credits for VBRMon.

```
Function:  Jump to address via JSR
Syntax:    J address
```

This function causes VBRMon to perform a JSR to some subroutine. The subroutine is executed within VBRMon's context, but the subroutine inherits (and modifies) the current registers (D0- D7/A0-A6) as displayed by the R command. The vector base register is set to 0 before jumping to the subroutine, and restored afterwards if the routine returns.

```
Function:  List hunt buffer
Syntax:    L
```

When the hunt (H) or offset hunt (O) command finds a match, it stores the address in a 36-entry hunt buffer, which may be displayed with this command. This avoids the need to repeat lengthy searches simply because the addresses have scrolled off the screen.

→

```
Function: Show memory (hex and ASCII)
Syntax:    M <start> <end>
```

The M command presents a simple hex and ASCII dump of memory. Note that the hex part is stored in the form of the memory modify command (:) so that you may edit the hex dump by simply moving the cursor to the byte you wish to modify, making the desired change, and hitting return.

```
Function: Toggle NTSC/PAL
Syntax:    N
```

This function switches Agnus (the 1MB or 2MB variety) from PAL to NTSC or vice versa. Note that VBRMon remembers your selection and forces either NTSC or PAL upon a warm boot, so it can do the job of the many "NTSC/PAL boot" programs in the public domain—and quite a bit more!

```
Function: Hunt for offset (PC-relative or branch) reference
Syntax:    O address
```

Often one wishes to search memory for a reference to a certain memory location (either code or data). Absolute references are easy to find; simply search for the destination address using the hunt command. However, PC (program counter) relative references (such as lea dest(pc),a0) and branch references (all of which are PC-relative) are more difficult to find. PC-relative references can be up to 32K away from the destination in either direction, so searching manually is out of the question. This command does it automatically, displaying the address of the instruction(s) that refer to the address given in the O command. The addresses are also stored in the hunt buffer (see the L command).

```
Function: Toggle printer output on/off
Syntax:    P
```

Enabling printer output sends all screen output to the internal parallel port (while still displaying it on the screen as usual). It works fine with my inexpensive Epson LX-80 printer, but is certainly not guaranteed to work with all printers. This function is extremely useful when needed.

```
Function: Show or modify registers
Syntax:    R<reg=value>
```

R by itself displays the current register set (usually the register set of the interrupted program) as well as INTENAR/DMACONR. The registers may also be modified, like so:

```
    rd0=123
    ra6=676
```

The modified registers will be passed to routines executed with the J command. In addition, your changes will affect the currently running program if you use the XMOD command to exit.

```
Function: Restore register set from memory
Syntax:    RESTORE address
```

The restore command retrieves a register set previously saved with SAVE and makes it the current register set.

```
Function: Step (find current track)
Syntax:    S drivenumber
```

The step command reports on what track (actually cylinder) the specified floppy drive is currently on. This is implemented by stepping the drive toward track zero, while examining the "track zero" input from the drive. The number of tracks stepped before reaching track zero indicates the current track. The state and position of the drive is restored afterward. Drive number ranges from 0 to 3.

```
Function: Save registers to memory
Syntax:    SAVE address
```

The SAVE function stores the current register set (plus program counter and INTENAR/DMACONR) at the indicated memory location. The layout is as follows:

```
    D0-D7      (long)
    A0-A6      (long)
    SSP        (long)
    USP        (long)
    SR         (word)
    PC         (long)
    INTENAR    (word)
    DMACONR    (word)
```

Altogether 78 bytes are used. The RESTORE command may be used to retrieve the registers.

```
Function:  Transfer memory
Syntax:    start end destination
```

The transfer command moves the memory region indicated by start and end (inclusive) to the destination address. A typical use is transferring the lower 512K to fast memory, as in T 0 80000 300000.

```
Function:  Triple compare
Syntax:    TC start1 end1 start2 start3
```

This is a special compare function that is typically used for finding magic locations in games. It compares corresponding bytes in three memory regions. If the first two memory regions have the same value, but the third memory region does not, then the change in value from the first two to the third is displayed in the following convenient form:

```
    Third_Region_Address: XX -> YY
```

An example is helpful in understanding why this is useful. Suppose you want to find the memory location containing the current number of lives in a game (Blood Money, let's say). The procedure is as follows. First, start the game and enter VBRMon. Copy the lower 512K to some free location. Then, do as much as possible in the game (move your ship around, shoot some enemies, etc.) without being killed. Again save the lower 512K to a free memory region. Now permit your ship to be destroyed. Assuming you have chosen to store the copies of the lower 512K in the first megabyte of autoconfig space, the appropriate triple compare command is as follows:

```
    TC 200000 280000 280000 0
```

In practice you will wish to perform the compare in stages, so as not to be swamped with information (if this happens, hit the escape key to abort). If you see something like "00008C29:03 -> 02", you've found the magic location!

```
Function:  Copper unassemble
Syntax:    U <start> <end>
```

This command implements a nice copper disassembler. Note that the disassembly process always halts when it encounters a CEND ($FFFF,$FFFE), a useful property.

```
Function:  Exit
Syntax:    X
```

The exit command restores the system state as much as possible and resumes execution. If the host application (e.g., game) reinitializes all the display registers in the copper list, the exit will be "clean." Otherwise the display will be mangled to varying degrees. There are two likely solutions to this problem (assuming you view it as problem):

1. Take advantage of an MMU to keep track of CPU writes to write-only custom chip registers. I only have a 68010; otherwise I would have done this.

2. Make a version of VBRMon that works over the serial port, much like RomWack. I may implement this in the future.

```
Function:  Exit with modified registers
Syntax:    XMOD
```

Same as exit above, but uses the modified register set (including SSP, USP, SR, and PC). See the R and J command descriptions for additional information.

```
Function:  Modify memory bytes
Syntax:    : address byte <byte> <byte> ...
```

This function simply places the specified series of bytes at the specified memory address. Strings are supported here as elsewhere (use single quotes). See the 'M' command description for an alternative method of modifying memory.

```
Function:  Calculate
Syntax:    ? expression
```

This command evaluates the given expression and displays the numeric result in hex, decimal, and binary. The available operators are as follows:

```
    +,-,*,/ - add, subtract, multiply, divide
    &  - and
    |  - or
    ^  - exclusive or
    ~  - not
    $  - hex override
    _  - decimal override
    %  - binary override
```

Multiple nestings of parenthesis are supported. The following precedence is maintained (from high to low):

```
1. Things in parenthesis
2. *,/
3. +,-
4. &,|,^,~
```

Please note that all commands support complex expressions of this form wherever a number is required. Text strings may be specified using single quotes.

Adding Your Own Commands

It is very easy to add your own commands, and I hope many of you will do so. To illustrate, let's examine one of the simpler commands already in VBRMon, the calculator:

```
Calculate:
          bsr          GetArg
          bcs.b        .EndCalc

;Print hex and decimal versions
          move.l       d0,-(sp)
          move.l       d0,-(sp)
          pea          (CalcFmt,pc)
          bsr          PrintLong
          lea          (12,sp),sp

;Print binary version
          move.l       d0,d1
          moveq        #31,d2
.binloop:
          move.b       #'0',d0
          lsl.l        #1,d1
          bcc.b        .skip1
          move.b       #'1',d0
.skip1:   bsr          PrintChar
          dbra         d2,.binloop
          lea          (MyCR,pc),a0
          bsr          Print

;Soak up excess args
..        bsr          GetArg
          bcc.b        ..
.EndCalc:
          rts

CalcFmt:
          dc.b         'Hex: $%081x',$a
CalcFmt1:
          dc.b         'Dec: %ld',$a
          dc.b         'Bin: %%'
          dc.b         0
          even
```

Note that this routine is not responsible for the grunt work of evaluating the given expression; rather, that is handled by the GetArg subroutine. GetArg will parse the command line starting with the current location and return a single number in D0.L indicating the final result of the expression. If the carry flag is set after calling GetArg, no

argument was available (end of command line). Strings are automatically handled. The calling program receives each character of the string as a separate argument. To retrieve a single raw character, call GetChar or _GetChar (the latter provides for register saving and other niceties).

Several routines are available for printing to the screen (and the printer if that option is enabled). Print takes a pointer to a simple text string in A0. PrintChar prints the character in D0.B. PrintLong is a Printf-like routine that accepts its arguments on the stack; see the above code excerpt for a usage example.

The preferred method of data storage is to allocate space in the global data section, which is always (by convention) pointed to by A5. Simply add the appropriate label and SO (structure offset) directive to reserve space, then reference it in your code as (MyData,a5).

Note that VBRMon was designed to be assembled with an assembler supporting the new Motorola syntax, such as Macro68. Please do not consider converting it by hand to the old syntax; it would just be an incredible waste of time.

The Credits

If I had had to write all of VBRMon from scratch I would not have even attempted it. Fortunately, the Amiga has developed a large software base from which to draw. I offer my sincere thanks to the following fine people: Dan Zenchelsky, for his expression evaluator, great late-night hacking sessions, tool development, and other contributions; Andreas Hommel for the mini-assembler; John Veldthuis for his nice copper disassembler; Dave Campbell for his nicely commented version of RomWack; and Swiss Cracking Association for the core program.

A Call to Hack

I hope you can appreciate the power VBRMon gives you to explore the Amiga. It has become a vital part of my programming and hacking toolkit. For the future, I'd like to add more debugging support (to further support the debugging of user-written games and demos), more commands, serial port support, MMU support, virtual 68000, etc. I can't do it alone, however. Take VBRMon as a starting point and then create your own masterpiece! ☑

About the Author

Dan Babcock is an electrical engineering major at Pennsylvania State University and an avid assembly language programmer. Contact him on People/Link as DANBABCOCK or on the Internet as d6b@ecl.psu.edu.

TO

> **" A way to automatically change the current directory based on a wildcard name. "**

The Development of an AmigaDOS 2.0 Command Line Utility

by Bruno Costa

"To" provides the user a way to automatically change the current directory based on a wildcard name, e.g., "to anim#?" might change directly, from any other directory, to a directory named ":Projects/Sculpt/Logo/Animation". A smart reader might be asking what reason was strong enough to make a programmer wait for AmigaDOS 2.0 to develop such a program. Elementary, my dear reader, although it was possible to write such a program under 1.3, it was necessary to fiddle with some AmigaDOS structures directly and this made things more complicated than desirable. Also, the excellent wildcard routines that are available under AmigaDOS 2.0 would have to be written under AmigaDOS 1.3, and that would make things much more complicated. As I examined the 2.0 include files, especially the pattern matching and process control routines, the idea of returning to that old project came to mind, and the result is now very useful but still simple enough.

This article has three main purposes, and three corresponding audiences: to someone who simply wants to install and use the utility, it will explain how the utility works from the user point of view; to programmers in general, it will explain how the utility is implemented, what are the algorithms and ideas behind it; and, last but definitely not least, to Amiga system programmers, it will explain how to use some of the new powerful features of AmigaDOS 2.0.

User Perspective

"To" is obviously a command line oriented utility - it should not be run from WorkBench since nothing useful will happen. Also, floppy disk users will not benefit from its use as much as hard disk users will, since this program saves more typing as the size of the directory trees grows — but, of course, nothing prevents floppy owners from using it. The program can be used at the same time in multiple hard disks, multiple partitions and multiple floppies.

This program is normally used with just one argument, a wildcard name that will be tested against all directories on disk. The current directory will be changed to the first directory which name matches with the wildcard. Note that just the directory name is considered for the search, e.g., if you have a directory called "Utilities:Comm/VT100" just "VT100" will be used. If there are multiple matches of a wildcard in the table, repeatedly calling "To" with the same wildcard will change to each match in turn, cycling back to the first when they are over. As an example, if you have just three directories that match with the wildcard "t#?", named "RAM:test", "RAM:t" and "RAM:clip/timing" and you type "to t#?" four times, you would change to "RAM:test" then to "RAM:t" then to "RAM:clip/timing" and finally back to "RAM:test".

"To" works based on a table that is written to the file "dir.table" at the root of every disk it is used in. If this table does not exist or is unreadable for some reason, "To" will try to rebuild it. This process will take some time on big disks, but can be safely interrupted at any time by pressing Ctrl-C. "To" has only one command line option, "-b", that forces the program to build the directory table for the disk from which it has been run. This option is to be used after creating or deleting some directories in the disk, and you want the new directory structure to be correctly handled by "To".

Technical Overview

Essentially, "To" does just two things: one is to gather data and build a table and the other is to read that table and use the data. The search is implemented by consulting a pre-computed table to obtain maximum efficiency, since it takes a considerable amount of time to read all directories in a disk - even the faster ones.

The table building part is based on a procedure that reads each directory, starting from the root, and calls itself recursively to read the sub-directories. For each directory it finds, this routine writes an entry in the table with just the full path name of this directory and some binary indexing information. Efficiency is the single reason for a binary table file to be used, instead of a simple ASCII one. In fact, if the table file was plain ASCII it could be built by the DOS command LIST, simply typing "LIST >:dir.table lformat %s%s ALL". The problem is that the program is used to build the table once (or twice) but reads it hundreds of times, so it is a better programming practice to make the reading process faster, even if this implies a more complicated or slower writing process.

After all directories have been read, the program writes at the start of the table file the number of directories that were found. This is done to simplify the reading process even further, since it allows the use of a fixed size structure (an array) instead of a variable size structure (a list), because even before reading the table, the program will know how many entries it will have to store. Note that this permits us to allocate all the memory at once - one big chunk - instead of many small allocations, and also avoids the memory and time overhead of linking list nodes.

The table reading part simply loads the table file in memory, and sets up two tables of the directories: one containing the full path name of each directory and the other containing just the name of each directory. The table with the names is used during the search, and the table with the full path names is used to change to a certain directory.

When a pattern matches with multiple entries in the table, the behavior described above is very convenient to the user, permitting him to cycle between the options using the same arguments to the command. To implement this, the program must have a way to know which one is the next option to be selected. An environment variable could be used here to indicate which was the last option selected, and from the value of that variable the program would determine the next one. This is not a bad idea in itself, but in this particular case there is already a "variable" indicating for us the last option chosen: the current directory name. This string, easily obtained from the system, will help the program to decide which is the "next" option in the user point of view. This is implemented by searching first for the current directory in the table and, after its place is found, the program searches from there on for a match with the pattern. The search continues in a cyclic fashion, going back to the start of the table when it reaches the end, until it finds a match or comes back to the current directory. If any match is found, "To" changes to the new directory.

You should notice at this point that the table is searched linearly, i.e., the program sequentially tests each entry in the table to see if it matches the pattern — if it does not, the program tries the next. It should be fairly obvious that this is not an efficient solution, since the program will have to test many entries until it finds the right one (or quits searching). There are at least two simple, efficient and well-known algorithms to search for names in tables: binary search and hashing. Both of these rely on a table previously ordered based on a particular criteria, and then start searching directly at a place that is usually very near to the target. This place is quickly calculated based on some properties of the string we are trying to find, and, because of this, both of the algorithms are inadequate for our use. Unlike these algorithms, described in many computer science textbooks, we are not searching for an exact match of two strings of plain characters — we are searching for a wildcard match. This makes the task of calculating an approximate guess based on the search string completely nonsense (at least using the traditional methods for the two algorithms mentioned), and binds us irremediably to the linear search. Since our main concern is execution speed, if the linear search is "fast enough" we won't have any reason to complain. Casual tests have shown that it has a good performance indeed: even with almost three hundred directories in a hard disk, "to" executes in a (sometimes lethargic) blink of the eyes, indistinguishable from the time taken to execute "CD", for instance ("CD" is obviously much faster, but the difference is not really noticeable to a human being).

AmigaDOS 2.0 Usage

Most of the code described up to now is relatively independent of AmigaDOS. If fact, the same ideas can be successfully used to implement a similar utility under other operating systems, such as MS-DOS and UNIX, if the system dependent functions are provided.

Some features of previous versions of AmigaDOS were used where there was no need for new functions. Some of the file managing routines, such as Open( ), Close( ), Read( ), Write( ), Lock( ), UnLock( ) and Seek( ) were used to handle the table file. IoErr( ) was used to handle AmigaDOS errors, Examine( ) and ExNext( ) were used to read directories and CurrentDir( ) was used to change the current directory. To better understand the usage of these functions, please refer to "AmigaDOS for Programmers" (AC's TECH premiere issue).

Two features of AmigaDOS 2.0 were specially important: the SetCurrentDirName( ) function and the pattern matching functions ParsePattern( ) and MatchPattern( ).

The SetCurrentDirName(name) function is used after you use CurrentDir( ) to change the directory, since the latter function does not change the directory name in the process control structure - just the current directory itself (odd, isn't it?). Previously, to achieve the same effect, you would have to change a field in the process structure by hand, and this involved freeing a memory area and allocating a new one using some memory managing conventions internal to AmigaDOS. Also, the current directory name is stored in the process structure as a BCPL string, which is not really practical to C programmers. There is also its counterpart function, GetCurrentDirName(buffer,length) that fills the string buffer with up to length characters of the current directory name. The addition of these two functions was very convenient to the development of "To".

The pattern matching facilities of AmigaDOS 2.0 are a key feature to make this program possible. Although the ARP (AmigaDOS Resource Project) library provided almost the same functionality as AmigaDOS patterns, there was at least one minor bug related to patterns, and ARP was not in such a wide use to assure that everyone would be able to use programs developed for it (I should point out that ARP was a remarkable effort, apparently with substantial influence in the development of AmigaDOS 2.0). There are two basic steps to use patterns under AmigaDOS 2.0: "compile" an ASCII pattern string into a more efficient representation, and test a pattern representation against an arbitrary string for a match. The former step is accomplished by ParsePattern(pattern, buffer, size), where pattern is simply a string that may include all wildcard constructions supported by AmigaDOS 2.0:

c	matches with character c
(chars)	matches with any character inside the brackets
(^chars)	matches with any character not inside the brackets
(str1 | str2 | ...)	matches with any of the strings str1,str2,....
#p	matches with zero or more instances of p
~p	matches with anything that does not match with p
?	matches with any single character
*	matches with zero or more characters of any kind (see the "WildStar" article, elsewhere in this issue of AC's TECH)

ParsePattern( ) fills the buffer (up to size bytes) with a compiled representation of the pattern, and returns TRUE if the string effectively contained wildcard characters. If it returns FALSE it means that you can safely use strcmp( ) or similar routines instead of MatchPattern( ), although the latter will still work. The length of the compiled representation is usually less than the original pattern, so you can safely use a buffer with the same size as the pattern string. If the buffer is smaller than needed, the routine will return -1.

To test a compiled pattern against an arbitrary string use the MatchPattern(pattern, string) call, that will return TRUE if the compiled pattern matches the given string under the rules described above. Note that this is what is called an "anchored"

match: each character of the string must match to a part of the pattern. This kind of pattern matching is very convenient to file names, but "free" matches are also useful in the context of text editors or text search utilities like the AmigaDOS search command. In this case, if any subset of the line matches the pattern, the whole line matches the pattern. This behavior can be simulated, with a tolerable performance, by calling MatchPattern( ) sequentially for each substring of the line, as in the following piece of code:

```
int FreeMatch (UBYTE *pattern, char *line)
{
  int i;

  for (i = 0; i < strlen (line); i++)
    if (MatchPattern (pattern, &line[i]))
      return TRUE;

  return FALSE;
}
```

It should be mentioned that there is a different set of pattern matching functions specifically designed for wildcard filename expansion (as in dir *.c), which should be used for that case. The MatchFirst( ), MatchNext( ) and MatchEnd( ) functions automatically compile a pattern, read all the required directories and return the proper path names of the matching files, if any. However, a complete description of their usage is out of the scope of this text.

AmigaDOS provided a simple but efficient set of file handling functions, basically formed by Open( ), Close( ), Read( ) and Write( ). These routines implemented what is called un-buffered files, i.e., each time Read( ) or Write( ) is called the physical device is accessed to get the information. When using buffered files, however, the information to be exchanged with the physical device is kept in a buffer managed by the system. This kind of I/O is mostly useful when the file is accessed in multiple small reads or writes. After multiple write operations, for instance, only after the buffer becomes full a real write access to the physical device will occur, writing all the information in a single big operation. Although buffered files are almost a must in these cases, the addition of this capability to AmigaDOS 2.0 will not be easily noticeable to most C programmers: the standard C I/O library provides this functionality (using fopen(), fclose(), fwrite( ), fprintf( ), fputc( ), fread( ) and so on) in a portable fashion, very easily implemented in the Amiga using Read( ), Write( ) and a bit of C code.

"To" uses some of these new buffered file handling functions, that are completely analogous to their standard I/O library lowercase counterparts. Most of these should be clearly described in your C compiler manuals or any C reference book, but some AmigaDOS details should be explained. The buffered files are opened exactly the same way as the unbuffered files were: using Open( ) and Close( ). To indicate that a particular file is to be buffered, you simply call SetVBuf (file, bufmode, bufsize).

The size in bytes of the buffer to be used (allocated internally by AmigaDOS) is given by bufsize, and bufmode is used to indicate which buffering mode is to be used: either BUF_FULL, BUF_LINE or BUF_NONE. BUF_FULL indicates that the buffer should be flushed only when it is absolutely necessary, BUF_LINE forces the buffer to be flushed whenever an end-of-line ("\n") is found and BUF_NONE disables any kind of buffering.

As in the buffered file case, this new version of Amiga-DOS also tried to offer to the programmer some more useful functions that had to be implemented over other basic primitives of the operating system. Some of them, especially the file name manipulation functions, have been used in "To". FilePart (fullpathname) returns a pointer to the file name part of fullpathname excluding the path, e.g., FilePart ("ram:t/test") returns "test". AddPart (fullname, objname, fullnamesize) concatenates objname (usually a directory or file name) to fullname (usually a pathname) adding a slash between them if needed. The resulting path name will use at most fullnamesize characters of fullname.

Another useful addition is the function NameFromLock (lock, lockname, namesize). It returns in the string lockname at most namesize characters of the full path name (including volume name and all directories) of the file or directory given by lock. As you should know, a lock is a handy unique identification for a file or directory, and it ensures that this particular object will be valid while you have it locked. Also, if the user renames a file, all locks to it remain valid, since they are independent of the file effective name. NameFromLock( ) may be seen as the inverse of Lock( ): Lock( ) returns a lock when given a file name, and NameFromLock( ) returns a file name when given a lock. Please note that this mapping is not necessarily one-to-one, since there are many path names by which a particular file may be referenced (using relative paths, assigned names and so on). To better understand NameFromLock( ) you might wish to examine an implementation of a similar function, given in "AmigaDOS for Programmers" with the name fullpath( ).

A new call has been provided in AmigaDOS 2.0 to check for a break request (either typing Ctrl-C, D, E or F, or issuing the shell command BREAK): CheckSignal (sigmask). It is essentially calls the exec function SetSignal( ) and checks if one of the signals in sigmask has been received; it was added just to convenience the AmigaDOS programmer. In the same sense, ExamineFH (file, fib) is exactly like Examine( ) but instead of giving the file by a lock, it is given by a file handle.

What if I don't have Workbench 2.0?

This is a very important question. It seems that the upgrade to Workbench 2.0 is not a trivial task to most Amiga owners, and some of them may prefer to postpone this upgrade to a later date. Specially in this conversion phase it becomes very important to try to support both systems, but since Workbench 2.0 is much more powerful, it will be probably difficult, most often impossible, to do so. If you run a program carelessly designed for Workbench 2.0 systems under 1.3, it will usually crash the machine immediately. Also, if you want to support both systems, it will be important to detect when Workbench 2.0 is available to be able to safely use its features. So it is now absolutely

necessary to check the correct versions of the libraries required. This is done in "To" by the the pair of functions DOSopen2_0( ) and DOSclose2_0( ). You should not underrate this warning, since this time, unlike the previous upgrades, the older version was not made obsolete — both are active and in use.

A Digression on File System Links

AmigaDOS 2.0 brings to the Amiga the concept of file system links, already known by UNIX users. Simply put, a link is a way to have two or more files, each with its own name, path and some attributes, referring to just one set of data. This means that just one copy of the file will be taking up space on disk, but to most (if not all) programs it will seem that each link is effectively a different file. Conceptually, there are two types of link: soft and hard. Hard links are usually indistinguishable from regular files, since they are built by making two file header blocks on disk point to the same set of data blocks. This construction is not possible if the files are in different file systems (different partitions of a hard disk or different devices). Soft links are simply a named reference to another file: a special file header block says something like "if this file is referenced by any means, use DH0:dir/file instead" (they are somewhat similar to a non-binding assign, as the PATH option of the 2.0 ASSIGN command). Note that the file is referenced by its path name, so files may be linked across devices and, in fact, the target file might not even exist. Both kinds of link are supposed to be implemented in AmigaDOS 2.0, but as the documentation warns, "soft links are not fully supported yet".

Sample Use of Links

"To" uses links just for tutorial purposes — they were neither necessary, nor even specially useful, in this simple application. Generally speaking, links can be used to create aliases for files. For example, a link can be used to create an alias for a command, as the ALIAS command does:

```
makelink C:ls C:list
alias ls list
```

The lines above produce essentially the same result, although using completely dissimilar techniques. In this particular case, the ALIAS solution should be used, since it is much more efficient and does not use any space on disk. There are situations, however, when ALIAS cannot be used and MAKELINK can: to create aliases for data files. If, for instance, a certain program requires a data file to be present in the current directory, instead of copying that file to every directory where you ran that program from, you could make links to a single copy of the data.

Links have been used in UNIX systems in both ways, even as command aliases, and there is a good reason for that. The ALIAS command is a feature of the shell, and it converts the alias name to the real program name before executing it — the program will never know that it was called by its alias. With links however, using the standard C startup, argv[0] will contain the name by which the program was called. This is useful if the program wants to assume a slightly different behavior when it is called by different aliases, effectively appearing to be distinct programs.

As an example, both the DIR and LIST commands could be links to the same executable, since both do the same thing (read directories) but print their results in a somewhat different format and support different options.

Since "To" has two distinct behaviors (building a table and using it), it is a valid candidate to use this command alias feature: if it is called by the name "mktotbl" (for "make the To table") it will behave as the table builder. To check which was the name by which it was called (either "To" or "mktotbl") it uses the AmigaDOS 2.0 GetProgramName (progname, namesize) call, that fills the string progname with at most namesize characters of the program name. It would be obviously possible to use argv[0] instead (in fact the startup code uses something similar to GetProgramName( ) to fill argv[0]). Note that in this particular case, the following line would have approximately the same effect as the link:

```
alias mktotbl "to -b"
```

As I mentioned before, this feature was added just as an instructive experience.

Source Code Guide

Apart from the error handling details, "To" will load the table, search for the current directory in the table, search from there on for a pattern match and if a match is found, change to the new directory. If the program was invoked with the "-b" option or by the "mktotbl" name, it will instead build the table.

The table construction is implemented by scandir( ), a recursive function that reads the contents of a given directory, adds each sub-directory to the table using addtotable( ) and calls itself recursively - non-directory entries are ignored.

The table saving process is composed by opentable( ), closetable( ) and addtotable( ): opentable( ) opens the file and writes the header; closetable( ) writes the directory count and close the file; addtotable( ) writes the given directory to the file and increments the directory count. These three functions and the scandir( ) function are called through the front end buildtable( ), that implements the proper calling sequence and parameters to effectively build the table.

Most of the reading table procedure is contained in loadtable( ). This function will try to open the table file, call readheader( ) to read the header and alloctables( ) to allocate memory for the tables with the size specified in the header. The table itself is read and stored in filebuf by a single Read( ) — if some kind of error occurs, loadtable( ) will call freetables( ) to free the memory. After that, two indexes for the table are built in memory, in such a way that there is an entry in each of them for each directory in the table: nametbl[n] points to the name and pathtbl[n] points to the full path name of the nth directory.

There are also the support functions, each one relatively simple and self-contained: fullname( ), used to get the full path name of a directory given by its name and a parent directory lock; cd( ), that changes the current directory to another directory given by name; and filesize( ), that returns the size in bytes of an opened file.

Examine the source code in the listing below carefully, as it is fully commented and explains some details that could not be mentioned here. You are also encouraged to modify it as you wish — as long as you do not redistribute the modified program — and use parts of the code anywhere you want. It should be mentioned that this small utility has been in use for a significant amount of time, and it is most unlikely that any serious problems may arise. Anyway, besides being a useful program, I hope that "To" will serve as an enjoyable learning tool.

Listing One

to.c

```c
/*
 * to.c - changes to any directory based on pre-computed directory tree
 *
 * Bruno Costa - 30 Jan 91 - 18 Feb 91
 *
 * (Requires AmigaDOS 2.0)
 */

#include <exec/types.h>

#include <dos/stdio.h>
#include <clib/dos_protos.h>
#include <pragmas/dos_pragmas.h>
#include <clib/exec_protos.h>
#include <pragmas/exec_pragmas.h>

#include <ctype.h>
#include <string.h>
#include <stdlib.h>


#define MAXNAME 100      /* maximun size of full path name */
#define MAXPAT 50        /* maximun size of a parsed pattern */

extern struct Library *SysBase; /* exec.library base */
struct Library *DOSBase;  /* dos.library base */

char fullprogname[MAXNAME];     /* [path] name of the program being run */
char *progname;                 /* name of the program being run */

char *filebuf = NULL;           /* buffer where table file data is stored */
long bufsize = 0;        /* size in bytes of the above buffer */

char **nametbl = NULL;          /* array of strings with directory names */
char **pathtbl = NULL;          /* array of strings with full path names */
long tblsize = 0;       /* size of the above arrays */

BPTR tablefile;                 /* handle to write to the table file */

#define USAGE_MSG "\x1b[32mto\x1b[31m v2.1 - \xa9 1991 by Bruno Costa\n" \
"usage: \x1b[1mto\x1b[0m [-b | <pattern>]\n"

#define HELP_MSG \
" Changes to a directory which matches <pattern>. Note that <pattern>\n" \
"may not include a path specification.  The search is based on a\n" \
"pre-computed directory table that is saved to ':dir.table'. If this\n" \
"table does not exist, it will be automatically created; otherwise the\n" \
"program will be instantly executed. Using the  option -b  will force\n" \
"the reconstruction of the table for the current drive.\n"


/*
 * Name of the file that contains the table. For each volume (disk,
 * partition, etc.) where TO will be used this file must be present, and
 * will contain a listing of all the directories present in that volume.
 */
#define TABLE_NAME ":dir.table"

/*
 * Table file header definitions. FILE_ID is a small string present in
 * the start of the file to help avoiding confusions - it is a hint that
```

```
 * we are reading a file in the expected format.  ID_SIZE is the size in
 * bytes of this ID string.  HEADER_SIZE is the size of the header, which
 * contains the file ID and a long int representing the number of
 * directories listed in this file.
 */
#define FILE_ID "DIRT"
#define ID_SIZE (sizeof (FILE_ID) - 1)          /* discounts '\0' */
#define HEADER_SIZE (ID_SIZE + sizeof (tblsize))

#define SORRY_MSG "Sorry, you must upgrade to " \
                  "Workbench 2.0 to run this program.\n"


/*
 *  Names by which this program is expected to be run.  Based on the name
 * by which the user calls the program, it will behave differently.  This
 * is meant to be used in the following way:  the executable is copied to
 * your C: directory with the name CHANGER; then you make a soft link
 * named BUILDER to it; then you may call either command by the correct
 * name and the same code will be invoked, but behaving apropriately.  Note
 * that only one copy of the file will be taking up space on disk.
 */
#define BUILDER "mktotbl"
#define CHANGER "to"


/*
 *  Returns TRUE if the FileInfoBlock structure pointed to
 * by fib refers to a directory.
 */
#define isdir(fib) (fib->fib_DirEntryType > 0)

/*
 * A new C language control structure ...
 */
#define loop for(;;)

/*
 * Returns TRUE if two given strings are the same (case-insensitive).
 */
#define strsame(str1,str2) (stricmp (str1, str2) == 0)


/*
 * functions defined in this file
 */

/* miscellaneous */
unsigned char *fullname (BPTR dirlock, unsigned char *name);
unsigned char *cd (unsigned char *dir);
long filesize (BPTR file);
void strlower (unsigned char *str);

/* table writing */
int opentable (void);
void closetable (void);
int addtotable (unsigned char *path);
int scandir (unsigned char *dirname);
int buildtable (void);

/* table reading */
int readheader (BPTR file);
int alloctables (void);
void freetables (void);
int loadtable (void);

/* Error handling */
void DOSopen2_0 (void);
void DOSclose2_0 (void);
void puterr (unsigned char *msg);
void fail (unsigned char *msg);


/*
 *  This program has essentially three main parts.  The first is a table
 * writer: it scans all directories in a particular drive or partition and
 * saves all directory names to a table file in that drive.  The second is
 * a table reader: it reads the table file produced by the first part and
 * stores a representation of it in memory.  The third is the table
 * scanner, that simply consults the table in memory to find a directory
 * that matches a user specified pattern and changes to it.  The first and
 * second parts are below main(), in apropriately labeled sections, and the
 * third is inside main itself.
 */
void main (int argc, char *argv[])
```

```
{
DOSopen2_0 ();

GetProgramName (fullprogname, MAXNAME);
progname = FilePart (fullprogname);          /* ignore path */

if (strsame (progname, BUILDER))             /* behave as BUILDER */
  {
  if (!buildtable ())
    fail ("could not build table.\n");
  }
else                                         /* behave as CHANGER */
  {
  if (argc != 2)        /* if incorrect # of args, prints help */
    {
    PutStr (USAGE_MSG);
    PutStr (HELP_MSG);
    }
  else                            /* if one arg, do the trick */
    {
    if (strsame ("-b", argv[1]))
      {
      if (!buildtable ())
        fail ("could not build table.\n");
      }
    else
      {
      int found;
      char **p, **n, **q, **cdptr;
      char pat[MAXPAT];
      char cdname[MAXNAME];

      if (!loadtable ())
        {
        if (!buildtable ())
          fail ("could not build table.\n");

        if (!loadtable ())
          fail ("table file is corrupt!\n");
        }

      GetCurrentDirName (cdname, MAXNAME);

      /*
       * If we are at the root directory, we know that it is not in the
       * table - we don't need to search.  Otherwise, search for the
       * current directory in the pathnames table.
       */
      found = FALSE;
      if (cdname[strlen(cdname) - 1] != ':')     /* are we at the root ? */
        {
        for (n = nametbl, p = pathtbl, q = n + tblsize; n < q; n++, p++)
          if (strsame (*p, cdname))
            {
            found = TRUE;
            cdptr = n;
            break;
            }

        if (!found)
          puterr ("current directory is not in table!\n");
        }

      if (!found)
        {
        n = nametbl;            /* if the current directory is not in the */
        p = pathtbl;            /* table, set up things so that the search */
        q = n + tblsize;        /* starts from the beginning of the table. */
        cdptr = n;
        }

      /*
       * Prepare a case-insensitive search (since table is all lowercase)
       */
      strlower (argv[1]);
      ParsePattern (argv[1], pat, MAXPAT);

      /*
       *  Searches for a directory name that matches the pattern, starting
       * from the next place after the current directory.  If there are
       * multiple matches of the same pattern in the table, this assures
       * that the user will be able to cycle among them just by repeating
       * the same command.  Note that we use the table in a cyclic fashion,
       * going back to the start if we reach the end of it - the search
       * only stops after we come back to the current dir after finding no
       * other match.
```

```
        */
       found = FALSE;

       do
       {
         ++n, ++p;  /* these pointers always go along */

         if (n >= q)       /* close the circle */
         {
           n = nametbl;
           p = pathtbl;
         }

         if (MatchPattern (pat, *n))
         {
           found = TRUE;
           break;
         }
       } while (n != cdptr);

       if (!found)
         fail ("no directory matches\n");

       if (!cd (*p))
         fail ("could not change directory\n");
     }
   }
 }

 DOSclose2_0 ();

 exit (RETURN_OK);
}


/*————————  Table Writing  ————————*/

/*
 *   The table is saved in the following format:
 *
 *         name        size(bytes)  contents
 *         ——          ——           ——
 *         FileID      4            'DIRT' - identifies table file
 *         Ndirs       4            number of entries that follows
 * E +—
 * n |     PathLen     1            size of the following string
 * t |     Path        PathLen      FULL path name of a directory
 * r |     EndPath     1            0 - terminates the path string
 * y +—
 *         ...         ...          ...
 */


/*
 * Opens and prepares the table file for writing. Returns TRUE if
 * successful.
 */
int opentable (void)
{
 tablefile = Open (TABLE_NAME, MODE_NEWFILE);
 if (!tablefile)
   return FALSE;

 /*
  * Use fully buffered I/O to minimize file manipulation overhead.
  */
 SetVBuf (tablefile, BUF_FULL, 4000);

 tblsize = 0;

 /*
  *   Write a header to identify the file, followed by the total number of
  * directory entries that follows. Note that at this time this number is
  * not yet known, so we write a 0 to reserve space for it, and patch it
  * with the corrected value later.
  */
 FWrite (tablefile, FILE_ID, ID_SIZE, 1);
 FWrite (tablefile, &tblsize, sizeof (tblsize), 1);

 return TRUE;
}


/*
 * Finishes the writing of the table file.
 */
void closetable (void)
{
 if (tblsize == 0)  /* an empty table does not make any sense */
 {
   Close (tablefile);
   DeleteFile (TABLE_NAME);
   puterr ("no directories found!\n");
 }
 else
 {
   /*
    * Write buffers to disk, to ensure a correct Seek() operation.
    */
   Flush (tablefile);

   /*
    *   Rewind to a place near the start of the file, where we reserved
    * space for the count of directory entries, and patch it with the now
    * known value.
    */
   if (Seek (tablefile, ID_SIZE, OFFSET_BEGINNING) == -1)
     puterr ("seek error!\n");
   else
     FWrite (tablefile, &tblsize, sizeof (tblsize), 1);

   Close (tablefile);
 }
}


/*
 *   Add a particular directory, given by its full path name, to the
 * currently open table file. Returns TRUE if successful.
 */
int addtotable (char *path)
{
 int pathlen = strlen (path) + 1;

 strlower (path);  /* table is all lowercase to be case insensitive */

 /*
  *   Write a string containing the full path name of a particular
  * directory, including a terminating '\0' and preceded by a byte with its
  * length.
  */
 FPutC (tablefile, pathlen);
 FWrite (tablefile, path, pathlen, 1);

 ++tblsize;

 if (CheckSignal (SIGBREAKF_CTRL_C))  /* check for ^C abort */
   return FALSE;
 else
   return TRUE;
}


/*
 *   Recursive function to read directories. For each directory given by
 * its full path name, it reads the contents of it adding each of the
 * directories inside it to the currently opened table file and calling
 * itself recursively. Returns TRUE if successful.
 */
int scandir (char *dirname)
{
 int success = FALSE;
 BPTR lock;
 struct FileInfoBlock *info = malloc (sizeof (*info));

 if (!info)
   puterr ("out of mem!\n");
 else
 {
   lock = Lock (dirname, SHARED_LOCK);
   if (!lock)
     puterr ("could not lock directory!\n");
   else
   {
     if (!Examine (lock, info))
       puterr ("could not examine directory!\n");
     else if (isdir (info))
     {
       loop
       {
         if (!ExNext (lock, info))
           break;                       /* if error, stop reading */
         else if (isdir (info))
```

UPDATE

Some readers have shown some concern about the unlharc'ing of the files on the AC's TECH disks. It seem that they get confused with the operation of lharc—especially if they are new to programming the Amiga.

Paul Castonguay has taken the initiative to write an extract program which will allow lharc file to be extracted from the WorkBench. We have implemented Extract starting with this diskette!

To extract an lharc file from the workbench:

*Lharc files will have a 'lock' icon—
Double-click the icon, which will run the
Extract application.*

A window will appear, asking you for a destination path.

Type your destination path (e.g. RAM: or dh0:) and a new drawer with the contents of the archive, will be created in the destination path.

Many thanks to Paul Castonguay for this fine addition to the AC's TECH Disk!

If you have any suggestion for AC's TECH, please send them to:

AC's TECH Suggestions
P.O. Box 869
Fall River, MA 02722-0869

AC's TECH Disk
Volume 1, Number 3

A few notes before you dive into the disk!

• You need a working knowledge of the AmigaDOS CLI as most of the files on the AC's TECH disk are only accessible from the CLI.

• In order to fit as much information as possible on the AC's TECH Disk, we archived many of the files, using the freely redistributable archive utility 'lharc' (which is provided in the C: directory). lharc archive files have the filename extension .lzh.

To unarchive a file *foo.lzh*, type *lharc x foo*
For help with lharc, type *lharc ?*

(see UPDATES for more information about unarchiving)

AC's TECH DISK GOES HERE!

Please notify your retailer if the AC's TECH Disk is missing.

We pride ourselves in the quality of our print and magnetic media publications. However, in the highly unlikely event of a faulty or damaged disk, please return the disk to *PiM Publications, Inc.* for a free replacement. Please return the disk to:

AC's TECH
Disk Replacement
P.O. Box 869
Fall River, MA 02720-0869

Be Sure to Make a Backup!

CAUTION!

Due to the technical and experimental nature of some of the programs on the AC's TECH Disk, we advise the reader to use caution, especially when using experimental programs that initiate low-level disk access. The entire liability of the quality and performance of the software on the AC's TECH Disk is assumed by the purchaser. PiM Publications, Inc, their distributors, or their retailers, will not be liable for any direct, indirect, or consequential damages resulting from the use or misuse of the software on the AC's TECH Disk. (This agreement may not apply in all geographical areas)

Although many of the individual files and directories on the AC's TECH Disk are freely redistributable, the AC's TECH Disk itself and the collection of individual files and directories on the AC's TECH Disk are copyright ©1990, 1991 by PiM Publications, Inc, and may not be duplicated in any way. The purchaser however is encouraged to make an archive/backup copy of the AC's TECH Disk.

Also, be extremely careful when working with hardware projects. Check your work, twice, to avoid any damage that can happen. Also, be aware that using these projects may void the warranties of your computer equipment. PiM Publications, or any of it's agents, is not responsible for any damages incurred while attempting this project.

```c
      {
        char *name = fullname (lock, info->fib_FileName);

        if (!addtotable (name))
          break;

        if (!scandir (name))
          break;                    /* if error, stop reading */
      }
    }
    if (IoErr () == ERROR_NO_MORE_ENTRIES)    /* if terminated normally */
      success = TRUE;

    UnLock (lock);
  }
  free (info);
}

return success;
}


/*
 *  Builds  the  directory  table  for  the  drive  where this command was
 * executed from.   Calls  scandir() to do the real work. Returns TRUE if
 * successful.
 */
int buildtable (void)
{
  int success;

  puterr ("building table - please wait ...\n");

  if (success = opentable ())
  {
    success = scandir ("*:");
    closetable ();
  }

  return success;
}


/*————————  Table Reading  ————————*/

/*
 *  Reads  the  header  of a table file, storing the useful information in
 * the  proper places.  It also checks if the file is in a suitable format.
 * Returns TRUE if successful.
 */
int readheader (BPTR file)
{
  char id[ID_SIZE + 1];

  bufsize = filesize (file) - HEADER_SIZE;
  if (bufsize > 0)
    if (Read (file, id, ID_SIZE) == ID_SIZE)
    {
      id[ID_SIZE] = '\0';
      if (strcmp (id, FILE_ID) == 0)
        if (Read (file, &tblsize, sizeof (tblsize)) == sizeof (tblsize))
          if (tblsize > 0)
            return TRUE;
    }

  return FALSE;
}


/*
 *  Allocates  the  tables  and  the buffer used to hold the table file in
 * memory. Returns TRUE if successful.
 */
int alloctables (void)
{
  filebuf = malloc (bufsize);
  nametbl = calloc (tblsize + 1, sizeof (char *));
  pathtbl = calloc (tblsize + 1, sizeof (char *));

  if (filebuf && nametbl && pathtbl)    /* if allocation succeeded */
    return (TRUE);

  freetables ();
  return (FALSE);
}
```

```c
/*
 *  Frees the tables and the buffer used to hold in memory the information
 * present in the table file.
 */
void freetables (void)
{
  if (filebuf)
    free (filebuf);
  if (nametbl)
    free (nametbl);
  if (pathtbl)
    free (pathtbl);
}


/*
 * Loads a table file in memory.
 */
int loadtable (void)
{
  BPTR f = Open (TABLE_NAME, MODE_OLDFILE);
  if (!f)
    puterr ("could not open table file\n");
  else
  {
    if (!readheader (f))
      puterr ("table is corrupt!\n");
    else
    {
      if (!alloctables ())
        puterr ("out of mem!\n");
      else
      {
        if (Read (f, filebuf, bufsize) != bufsize)
          puterr ("error reading table\n");
        else
        {
          int i = 0;
          char *p = filebuf, *q = p + bufsize;

          /*
           *  Set up the arrays of directory names and path names to point
           * to  the  proper places in the  file buffer, which contains
           * essentially a copy of the file contents.
           */
          while (p < q  &&  i <= tblsize)
          {
            pathtbl[i] = p + 1;
            nametbl[i] = FilePart (p + 1);
            p += *p + 1;
            ++i;
          }

          if (p == q)            /* if the loop terminated normally */
          {
            Close (f);
            return TRUE;         /* if successful, return here */
          }
        }
        freetables ();
      }
    }
    Close (f);
  }
  return FALSE;
}


/*————————  Miscellaneous  ————————*/

/*
 *  Returns  a  pointer  to a static buffer (overwritten by each call)
 * containing  the  full  path name of file with a given name relative to a
 * given lock.
 */
char *fullname (BPTR dirlock, char *name)
{
  static char fullname[MAXNAME];

  NameFromLock (dirlock, fullname, MAXNAME);
  if (name)
    AddPart (fullname, name, MAXNAME);

  return fullname;
}
```

```
/*
 *  Changes  the  current  directory  to  the  one given by its path name.
 *  Updates  the  proper  Process structure fields and returns the full path
 *  name of the directory.
 */
char *cd (char *dir)
{
BPTR newdir, olddir;
static char namebuf[MAXNAME];

newdir = Lock (dir, SHARED_LOCK);
if (!newdir)
  return NULL;
else
  {
  olddir = CurrentDir (newdir);
  if (olddir)
    UnLock (olddir);
  NameFromLock (newdir, namebuf, MAXNAME);
  SetCurrentDirName (namebuf);
  return namebuf;
  }
}


/*
 * Returns the size in bytes of a file given by a handle.
 */
long filesize (BPTR file)
{
long size = -1;
struct FileInfoBlock *fib = malloc (sizeof (*fib));

if (fib)
  {
  if (ExamineFH (file, fib))
    size = fib->fib_Size;

  free (fib);
  }

return size;
}


/*
 * Converts all uppercase characters in a string to lowercase.
 */
void strlower (char *str)
{
for ( ; *str; str++)
  if (isupper (*str))
    *str = tolower (*str);
}
```

```
/*-------------  Error Handling  -------------*/

/*
 * Writes an error message, preceded by the program name.
 */
void puterr (char *msg)
{
PutStr (progname);
PutStr (": ");
PutStr (msg);
}


/*
 * Writes an error message, and then exits with a failure code.
 */
void fail (char *msg)
{
puterr (msg);
DOSclose2_0 ();
exit (RETURN_FAIL);
}


/*
 *  The two functions below are used  to  assure that AmigaDOS 2.0 is
 *  present  which  is  required for this program to work.  If it isn't, the
 *  code  below tries to open the previous version of the library (1.3) just
 *  to print a message, and exits.
 */
void DOSopen2_0 (void)
{
DOSBase = OpenLibrary ("dos.library", 36);  /* requires 2.0 */
if (DOSBase == NULL)
  {
  DOSBase = OpenLibrary ("dos.library", 33);      /* try opening 1.3 */
  if (DOSBase)
    {
    Write (Output (), SORRY_MSG, sizeof (SORRY_MSG));
    CloseLibrary (DOSBase);
    }
  exit (RETURN_FAIL);
  }
}


void DOSclose2_0 (void)
{
if (DOSBase)
  CloseLibrary (DOSBase);
}
```

✓

Programming the Amiga's GUI in C

PART II—MODULAR PROGRAMMING & AN INTRODUCTION TO WINDOWS

by Paul Castonguay

This is the second in a series of articles that will help you take advantage of many of the custom features of your Amiga using the C Programming Language. In this issue you will find:

1. A discussion of modular programming using the SAS/C, version 5.10A.
2. A presentation of how windows are opened in the Intuition programming environment.
3. Some background material (on disk & page 74) for understanding how to best configure your system.
4. A detailed discussion (on disk) of the installation programs that appeared in the last issue of AC's TECH.

Structured Programming, Modules

One of the most important concepts in structured programming is the idea that a project should be broken down into precompiled modules. There are two reasons for this. First, it makes the solution of a complex programming task easier to deal with by treating it as a combination of several, more fundamental and easier to understand sub-tasks. Second, it saves time by removing the requirement that an entire project be re-compiled every time a modification is made to only one small part of it, or that a particular module be re-compiled every time it is imported into a different project.

Last article we saw the development of a program (Libs.c) to open various "Run Time" libraries that we will use in most of our projects. We saw that it involved many issues, far too many to deal with every time we start a new project. We therefore want to develop a general solution that will be reusable in any new project.

One way you could use the above mentioned program is to join (or merge) the source code of the Open_Libs() and Close_Libs() functions into that of other projects. Although that would be easier than rewriting those functions from scratch, it would still require that they be compiled along with the other project. Compiling is a time consuming process and we should not waste valuable development time doing it over and over again to the same functions every time we want to use them in a different project. For that reason every development package (like SAS/C) allows for the creation of compiled versions of functions (or collections of related functions) that can be joined to other programming projects not at the source code level, but at the object code level, by the linker. I call these compiled functions modules.

Below I give the listing of Libs.c with the exception that this time there is no main() function. Recall from Standard C that every C program requires a function called main() (page 6 of "The C Programming Language", by Brian Kernighan & Dennis Ritchie, Second Edition, Prentice Hall 1988, referred to in this article as K&R). Thus it does not represent a complete program. It is only

a module. It is also available on the disk that comes with this magazine, so you don't have to actually type it in. Note that the version on disk contains extra comment lines not shown below that act as documentation.

```
/*                      Libs.c                      */
/*                                                  */

#include <intuition/intuitionbase.h>
#include <graphics/gfxbase.h>
#include <libraries/diskfont.h>
#include <proto/exec.h>
#include <stdio.h>

#define INTUITION_REV 33
#define GRAPHICS_REV 33
#define DISKFONT_REV 33


         /* === GLOBAL pointers required by system === */

struct IntuitionBase *IntuitionBase = NULL;
struct GfxBase *GfxBase = NULL;
struct Library *DiskfontBase = NULL;


              /* === Functions in this module === */

BOOL Open_Libs(VOID);
VOID Close_Libs(VOID);

BOOL Open_Libs(VOID)
{
    if(!(IntuitionBase = (struct IntuitionBase *)
                 OpenLibrary("intuition.library", INTUITION_REV)))
    {
        printf("Can't load intuition.library.\n");
        return(FALSE);
    }

    if(!(GfxBase = (struct GfxBase *)
            OpenLibrary("graphics.library", GRAPHICS_REV)))
    {
        printf("Can't load graphics.library.\n");
        Close_Libs();   /* close IntuitionBase */
        return(FALSE);
    }

    if(!(DiskfontBase = (struct Library *)
```

```
                OpenLibrary("diskfont.library", DISKFONT_REV)))
    {
        printf("Can't load diskfont.library.\n");
        Close_Libs();    /* close IntuitionBase & GfxBase */
        return(FALSE);
    }

    return(TRUE);
}


VOID Close_Libs(VOID)
{
    if(IntuitionBase)
        CloseLibrary((struct Library *)IntuitionBase);

    if(GfxBase)
        CloseLibrary((struct Library *)GfxBase);

    if(DiskfontBase)
        CloseLibrary(DiskfontBase);

}
```

We now want to compile this module, but not link it (see background material on compiling and linking last issue). I will do this first from the SHELL window, the icon method will follow shortly.

Copy the Libs.c file from the magazine diskette to your current working directory, either RAM:, RAM:My_Project, or SAS_Headers: depending on the size of your system and how you like to work. What follows does not take up much memory so don't be afraid to use RAM: on a 512K system.

```
Copy Source_Code/Libs/Design/SHELL_Version/Libs.c to ""
```

Notice the use of double quotes "" with nothing in between to copy a file into a current directory. This is a feature of version 1.3 AmigaDOS. Next enter the following command:

```
LC Libs
```

This produces a file called Libs.o, representing the object code of the above listing. It is now a compiled module.

Check Out the Size

Use the AmigaDOS List command to see the size of the Libs.o file. On my system I get 684 bytes. That's pretty small, smaller in fact than the original source code (Libs.c) that produced it. Now to help you gain a better understanding of what is going on here take a look at the size of the compiled and linked program that you produced last issue. On my system it was 7240 bytes. The reason for this large difference in size between a program and a module is the fact that the program has been linked. It has had added to it the routines required to run it on the system.

Just for the fun of it try running the above compiled module by entering the following command into a SHELL window:

```
Libs.o
```

My computer reports: "Unable to load Libs.o: file is not an object module". You see, a module does not represent a complete program. It is however in a compiled form, ready to be used in other programming projects. But before seeing how that is done I need to say a few words about compiler options.

Compiler Options

So far you have been merrily compiling and linking by entering the LC command in a SHELL window, or by double clicking the "Build" icon on the Workbench, without giving too much thought as to exactly how that job got done. It just so happens that the LC program has a series of default settings (options) that it uses in the absence of any given by you. But default options do not satisfy every programming situation. I therefore recommend that you learn a few important ones.

From the SHELL window you specify compiler options by entering them on the command line after the LC or LC1 command and before the name of the program you want to compile. You must also precede them with a hyphen, like this:

```
LC -Lm myprogram
```

See the background material in the last issue of AC's TECH/ AMIGA. Remember that options are case sensitive, -Lm is not the same as -LM or -lm.

LC's -c Option

The -c option is called the compatibility option and it is used to activate certain features having to do with ANSI compliance, as well as compatibility with earlier compilers. This option is used in conjunction with one or more other letters, each representing a different feature. Below I describe the ones that I will use.

-cf Option

I'm sure you have noticed the extra effort that I have taken to write my functions in the style recommended by the new ANSI standard (K&R, page 26). This makes it possible for the compiler to pick up errors having to do with the number and type of arguments used in function calls. It does that by comparing function calls against their prototype definitions, and to work properly every function in your program must have a prototype definition. The -cf option forces the compiler to verify the presence of a prototype definition for every function in your program and warn you if one is missing.

-ci Option

In a complex project the number of header files can be quite large, making it highly likely that you will accidentally #include a particular one more than once. The -ci options protects you against that by making the compiler suppress multiple inclusions of the same header file, using only the first one.

-cs Option

You should remember from Standard C that whenever you use a string constant in one of your programs the compiler must store it somewhere in memory. The statement:

```
printf("Hello World.\n");
```

→

causes the allocation of 14 contiguous (adjacent) bytes in memory containing the above string, complete with a '\0' to signal the end of the string. When the printf() function is executed it receives as input not the string itself, but the address in memory where it is located.

String constants are used often in programs and occasionally the same one is used more than once in the same program. Perhaps there are several printf() statements containing the same message, like this one:

```
printf("Error, program aborted!!\n");
```

The -cs option causes the compiler to take advantage of this and create only one string in memory to serve the several printf() statements that use it, giving to each the address of the same place in memory. It makes your program more compact.

-ct Option

A fact of life that you must deal with when programming in the Amiga's Intuition environment is the heavy use of structures (K&R, page 128) and pointers to structures. You have already seen that something as simple as a pointer to a library is actually a pointer to a structure, specifically of type "struct IntuitionBase". It turns out that the accidental use of undefined structures can occur often in your programs as a result of either forgetting a necessary header file, or by making a simple typo error, like entering "Intuitionbase" instead of "IntuitionBase" (lower case 'b' instead of upper case 'B'). The -ct option causes the compiler to warn you if your program attempts to use a structure before it is defined.

-cc Option

A common practice when debugging programs is to comment out certain parts, thus compiling the program with that part removed. It helps locate the exact location of bugs in your source code. However the compiler will get mixed up if the section of code you comment out has some comments within it. The -cc option allows the use of nested comments, comments within comments. It allows you to comment out any section of your program and not have to worry about whether or not there are comments within that section.

To use all the above features together you combine them in a single option field with the c as the first character and the rest in any order, like this:

```
LC -cfistc myprogram
```

-b1 Option, Relative Addressing

This option has to do with the way the MCS68000 microprocessor (the central processing unit inside the Amiga) specifies addresses in memory. It can do so either absolutely using a 32 bit number (ULONG), which can represent any address in memory, or relatively using a 16 bit number (WORD), which can represent addresses only as an offset from some other address. Relative addressing requires less machine code instructions, resulting in

faster executing programs. However, it has the disadvantage that it is not able to reach sections of memory outside a single 64K segment. We will use the faster executing relative addressing mode (option -b1) and in the event that our programs become too large, the compiler will warn us that it cannot reach all parts of it. At that point we will have to use the compiler option that specifies absolute addressing, which is -b0.

Combining all the above options together for our Libs.c example we have:

```
LC -b1 -cfistc Libs
```

or, calling the first and second pass compiler programs directly we have:

```
LC1 -b1 -cfistc Libs
LC2 Libs
```

Try the above commands from the SHELL window on the Libs.c module and confirm that the compiler produces a more efficient version occupying 656 bytes instead of the 684 produced earlier.

-o Option. LC1, LC2, And the QUAD: Device

The LC controlling program writes and reads the quad file (intermediate file produced by LC1) to and from the QUAD: device. However, the LC1 and LC2 programs, when called directly, do not recognize the QUAD: device. You may not want that. To force the LC1 and LC2 programs to recognize the QUAD: device (perhaps to conserve memory, perhaps for smoother operation) you must use another option, the -o option:

```
LC1 -b1 -cfistc -oQUAD: Libs
LC2 -oLibs.o QUAD:Libs
```

The above commands cause the quad file to be written to and read from the QUAD: device, and the object code (compiled module) to be placed in the current directory. Later in this article you will see that there is an easy way to deal with such long option fields.

Compiling a Module from Icons

If you have already tried to compile Libs.c by dragging it into a new project drawer (one created using SASCSetup) and double clicking the "Build" icon, you saw the system try in vain to link it as well. On a two floppy system it probably complained that it could not locate the file "amiga.lib". On a hard drive system the BLink program reported that _main was an undefined symbol. Be careful! Any attempt to run such a program will crash your machine.

Nothing is wrong here. What you are seeing is a result of the fact that the icon method of compiling, lacking instructions to do otherwise, tries to link all programs that it compiles. But since Libs.c does not represent a complete program (it has no main() remember?) the BLink program experiences difficulties with it.

The Lattice Make Facility

Please try the following little experiment. Single select the "Build" icon in any project drawer and then pick "Info" from the Workbench pull-down menu. The default tool is LC:LMK, not LC:LC as you might have expected. Thus double-clicking the "Build" icon invokes not the LC program but another one called "LMK". This program is usually referred to as "Lattice Make". Its purpose is to supervise the compiling and linking of your project. Let's try a short demonstration.

A Short Working Example, Using Icons

Copy the entire Source_Code/Libs/ICON_Version drawer from the magazine diskette to your current working directory, open it up, and double-click its "Build" icon. Libs.c compiles but this time no attempt is made to link it. Close the window and re-open it and you will see a new icon called Libs.o, your new module.

How Come it Worked?

The ICON_Version of Libs has a new file inside called "LMKFile". Double-click its icon and the LSE editor will show you its contents:

```
Libs.o : Libs.c
   LC -b1 -cfistc Libs
   Copy LC:Icons/def_exe.info to Libs.o.info
```

So it turns out that to compile a program you don't actually invoke the LC program directly. Instead you put instructions to do so in a file called LMKFile. Then the LMK program (invoked by double-clicking the "Build" icon) sees to it that those instructions get executed. Notice that within the LMKFile I specified the full set of compiler options that we discussed earlier. I will discuss the exact syntax rules for the LMKFile instructions later in this article. You may now quit the LSE editor.

The Power of LMK

Double-click the "Build" icon again. This time Libs.c does not get compiled. Instead LMK reports that your project is up to date. Here lies the real power of LMK. It does more than just execute instructions to compile your modules, it also determines whether or not those modules need to be compiled in the first place. In this case it determined that Libs.c did not need to be re-compiled. Now double-click the Libs.c icon, re-save it ([RIGHT-AMIGA]-S), and then double click the "Build" icon again. This time LMK decides that re-compilation is required.

LMK from the Shell

The LMK program is not just a trick to allow you to compile modules using icons, it is useful from the SHELL as well. Let's go back to the SHELL version and compile the module again, this time using the LMK program.

Delete the Libs.o module that you compiled from the SHELL earlier, then copy into that same directory all files from Source_Code/Libs/Design/SHELL_Version on the magazine diskette.

```
Copy Source_Code/Libs/Design/SHELL_Version to ""
```

Now enter:

```
LMK
```

and the module gets compiled. Here is the corresponding LMKFile:

```
Libs.o : Libs.c
   LC1 -b1 -cfistc -oQUAD: Libs
   LC2 -oLibs.o QUAD:Libs
```

Notice that in the above script I called the first and second pass compiler programs directly rather than through the LC controlling program. I did that in the interest of memory efficiency, for owners of 512K systems. Enter the LMK command a second time and it will report to you that your project is up to date.

It Isn't Magic

The LMK program relies on the date/time stamp of your files for its operation. You should now begin to appreciate why in the 512K installation instructions (last issue) I emphasized the importance of entering the correct date and time every time you use your computer. In the above example the LMK program compared the time stamp of Libs.c with that of Libs.o to determine whether or not Libs.c had been modified since the last compilation was performed.

LMK Helps You Organize Your Projects

A project can consist of many modules, each one dependent in some way on one or more of the others. A modification to one might require that certain ones be re-compiled, while others remain unaffected. Only you will know for sure because it is you who is designing the project. But as your project gets larger and the number of modules in it increases, even you will begin to loose track of exactly how they depend on each other. The LMK program can help you maintain control over such issues by following the instructions that you specify in the LMKFile. You write these instructions for each module at the same time that you design it, when you are most familiar with how it affects and interacts with other modules in the project.

Once the instructions for a particular module are written you can essentially forget about them, the LMK program will execute them as required whenever you enter the LMK command, or double-click the "Build" icon. If you design more modules you simply add more instructions to the LMKFile. Each time you invoke the LMK program it will compile only those modules which are new or have been modified. Naturally there are some syntax rules that you must learn in order to write instructions in the LMKFile, but first let's finish our module example.

Testing Our Module

You must always test a module before using it in other projects. To do that you write a driver program, a short main(), that calls the functions in the module. This is very similar to the example last issue except that this time the test program is in a different file than the functions, and the functions are in a compiled form.

Recall from standard C that if one part of your program wants to use a variable or function that is defined in another file you must use an "extern" declaration for it (K&R, page 80). Our module contains two functions, Open_Libs() and Close_Libs(), as well as three pointer variables that must be global in scope. Any project that wants to use this module must therefore make extern declarations to those functions and pointers. But wait a minute, isn't that a lot of work? After all, the whole idea of building the module in the first place was to protect us from exactly these kinds of internal details.

A common way of handling this complexity is to make a header file for the module. When you want to import the module into a new project you make all its required declarations by putting into your source code one simple #include statement to the module's header file. This is another example of the formal principle called "Information Hiding". Below I give you the header file, the test program, and the LMKFile that together link and test the Libs.o module. Note that the version of Libs.h on disk contains comments not shown below. These are detailed descriptions on how to use the functions in Libs.o, making it easier for you to do so at a later date. I'll have more to say about this next issue.

MODULE HEADER FILE

```
/*                        Libs.h                      */
/*                                                    */
/* AC's TECH/AMIGA                   Volume 1, Number 3 */


        /* === GLOBAL pointers required to use Libs.o === */

GLOBAL struct IntuitionBase *IntuitionBase;
GLOBAL struct GfxBase *GfxBase;
GLOBAL struct Library *DiskfontBase;


        /* === Functions within Libs.o module === */

GLOBAL BOOL Open_Libs(VOID);
GLOBAL VOID Close_Libs(VOID);
```

MODULE TEST PROGRAM

```
/*                     Test_Libs.c                    */
/*                                                    */
/* AC TECH                           Volume 1, Number 3 */

#include <stdio.h>
#include <stdlib.h>
#include <libraries/dos.h>
#include <proto/dos.h>

#include "Libs.h"

VOID main(int argc, char *argv[]);

VOID main(int argc, char *argv[])
{
  if(!Open_Libs())
  {
    printf("Trouble opening libraries\n");
    Delay(100);
    exit(RETURN_WARN);
  }

  printf("intuition.library = %u\n", IntuitionBase);
  printf("graphics.library = %u\n", GfxBase);
  printf("diskfont.library = %u\n", DiskfontBase);
```

```
  Delay(100);

  Close_Libs();

  exit(RETURN_OK);
}
```

TEST LMKFILE

```
Test_Libs : Test_Libs.o Libs.o
  BLink FROM LIB:c.o+Test_Libs.o+Libs.o TO Test_Libs LIBRARY LIB:lc.lib

Test_Libs.o : Test_Libs.c Libs.h
  LC -b1 -cfistc Test_Libs
```

You can find these files on the magazine diskette in the directory:

Source_Code/Libs/Test/SHELL_Version/

> *or*

Source_Code/Libs/Test/ICON_version/

depending which method you choose for using your compiler. Double-click the "Build" icon, or enter LMK in a SHELL window, and witness the compiling and linking of the project. You will see that the file Libs.c is not re-compiled. In fact, it isn't even present in that directory. Instead, the Libs.o module is linked directly into the project by the BLink program. Congratulations, you have just built your first pre-compiled module and linked it into a project, albeit a very small one.

You might be wondering about a few things in the above program. First of all I have changed the type for the function main() from previous examples. I am now following the form suggested in the ROM KERNAL manual, as well as the new ANSI standard. The function main() is defined as type VOID, meaning that it will return no value. To terminate the program I use the exit() function which transfers control back to the operating system and passes to it an integer value. To remain compatible with other AmigaDOS programs, I return the values defined in the macros RETURN_OK, RETURN_WARN, RETURN_ERROR, and RETURN_FAIL. These contain values of 0, 5, 10, and 20 as defined in the <libraries/dos.h> header file. Another change to main() is the use of the variables argc and argv for accepting command line arguments, even though my program doesn't actually make use of them. This is becoming the new accepted Standard C form for the main() function of any program.

The above changes require that I add a prototype declaration for main(). "Why not do that in a header file?", you ask. Well, if you look in <stdlib.h> (line 79) you will see that there is one there, but it has been commented out. If you remove the comments and re-install your header files you won't have to put a prototype declaration for main() in your programs, assuming that you #include the <stdlib.h> header file of course. But be warned that

your compiler may now complain if you try to compile programs that use the older integer type main() or that don't use the argc and argv arguments. Most C textbooks examples are still using the old form.

Each #include statement in the Test_Libs.c program is made for a different reason. <stdio.h> contains the prototype definition of the printf() function. <stdlib.h> contains the prototype definition of the exit() function. <proto/dos.h> contains the prototype definition of the Delay() function. And finally "Libs.h" contains the prototype definitions and global variable declarations for the Lib.o module. Notice that angle brackets <> are used to indicate header files that are stored in the INCLUDE: device, where system header files are located, and quotation marks "" are used to indicate header files that are stored in the current directory, where project header files are located.

The above example required that the Libs.o and Libs.h files be in the current directory. I put them there for you this time, but in the next example you will have to do that yourself.

You must now be careful to not loose the module Libs.o and its related header file Libs.h. Together these two files represent an easy to use, streamlined solution (in the form of a pre-compiled module) for opening the Amiga's "Run Time" libraries. You have worked hard to produce that module and you don't want to waste valuable time doing it again. In the future you will use it by copying the Libs.o and Libs.h files into the drawer (directory) of other projects. In the meantime you need a safe place to archive it. I recommend that you start a special library diskette to hold all your pre-compiled modules, as well as their related files.

Suggested File Structure

I said last issue that programming in C was largely a matter of organizing your work in such a way that it becomes usable in most programming situations. With that in mind the way you archive your work is very important. If you archive Libs.o and Libs.h alongside other files, like Libs.c and LMKFile, you may forget exactly which files you're supposed to copy into other projects in order to use the module. At the same time you must archive all files that were used in its design in the event that you want to modify it at some later date.

Figure One shows a suggested file structure for archiving the above module. Notice that the only files visible in the Libs directory are Libs.o and Libs.c, the ones you need to copy to other projects in order to use the module. Design and Test are directories. At the same time all support files needed for its design and verification are safely tucked away in their own respective directories, available for future program maintenance.

Making an LMKfile

Now that you have seen the LMK program in operation, and convinced yourself of the need to use it, you need to learn the syntax rules for writing instructions into the LMKFile. The main idea behind these instructions is to tell the LMK program what files to produce, when, and how. This is done by giving it a series of target files, dependent files, and actions.

➤

TARGET FILES: Files that you want LMK to produce

DEPENDENT FILES: Files that you want to affect the generation of the target files.

ACTIONS: Specific instructions for producing target files from their dependents.

Target files are specified as a name followed by a space, then a colon. They must be written starting in the first column (along left edge of editor window). In the Test_Libs example the files Test_Libs and Test_Libs.o are entered as target files. They are files that you want the LMK program to produce.

```
Test_Libs : Test_Libs.o Libs.o
    BLink FROM LIB:c.o+Test_Libs.o+Libs.o TO Test_Libs LIBRARY LIB:lc.lib

Test_Libs.o : Test_Libs.c Libs.h
    LC -b1 -cfistc Test_Libs
```

Dependent files are listed after the colon on the same line as the target file that you want them to affect and they are separated by spaces. Thus we have the form:

```
    target file : dependent file list
```

In the above example Test_Libs.c and Libs.h are dependent files of the target file Test_Libs.o. They are used to produce it. Similarly, Test_Libs.o and Libs.o are dependent files of Test_Libs.

Actions are specified on the line or lines immediately following the "target : dependent list" and are indented one tab. They contain specific instructions on how to produce a target from its dependent files. In the above example the line "LC -b1 -cfistc Test_Libs" tells LMK how to produce Test_Libs.o from Test_Libs.c and Libs.h. Note that the target file Test_Libs.o is itself a dependent file of Test_Libs and appears as such in the first instruction.

The order of appearance of each target file in the LMKFile is in reverse to how it is produced. The rule to remember is that no target should appear earlier in the file than another for which it is also a dependent. In our above example the first entry, Test_Libs, is not a dependent file of any other target and therefore appears before all others in the file. However Test_Libs.o is a dependent of Test_Libs. Therefore the target Test_Libs.o must appear later in the file than the target Test_Libs.

A blank line is use to separate each group of targets, dependents, and actions.

Date/Time Stamps

If a target file does not yet exist (as is the case when you are compiling a file for the first time) the LMK program executes its action lines. On the other hand if the target file already exists the LMK program checks the date/time stamp of all its dependents. If any dependent file has a date/time stamp later than that of the target (as would be the case if you made a modification to one of them since the last time the project was compiled) the LMK program will execute the action lines. But if the date/time stamps

of all dependent files are earlier than that of the target (meaning that no modifications were done since the last compilation) the LMK program will decide that the last compilation is still valid and it will not execute the action lines.

Look at the following "target : dependent" relationship (again from the above example):

```
Test_Libs.o : Test_Libs.c Libs.h
    LC -b1 -cfistc Test_Libs
```

The file Libs.h is listed as a dependent file of Test_Libs.o. I did that because the Test_Libs.c file contains an #include statement for Libs.h. Thus any modification to Libs.h could affect the target in some way. But Libs.h is not part of the action line. That's because it is #included in the Test_Libs.c file, getting processed automatically when Test_Libs.c is compiled. This example demonstrates how a project consisting of several modules, each with its own header file, can easily get out of control if modifications are made to certain files without properly re-compiling certain others. A change to Libs.h requires that Test_Libs.c be re-compiled and forgetting to do so would cause that change to not get incorporated into your project. As the number of modules in your project increases you may begin to loose track of such issues. The LMK program in conjunction with the instructions that you write in the LMKFile allows you to work on your project with a minimum of compiling and without having to think out the dependency of each module every time you make a modification to one of them.

Warning About Dragging Modules

Note that copying a project by dragging its drawer icon (on the Workbench screen) changes the date/time stamp of all its files at the new location. Thus a project that is up to date in your RAM Disk may no longer be so when you archive it to a diskette. If you later drag that project back into RAM: and double-click its "Build" icon the LMK program may decide to re-compile the whole thing all over again even though it isn't really necessary. But even more importantly you may have wanted to keep an accurate record of when modifications were last performed on each module file in your project.

Keeping a Date

To copy files without destroying their original date/time stamps you must use the Copy command from a SHELL window with the "clone" option. Note that Commodore has defined the xcopy command (as an alias name) to be equivalent to copy with the clone option.

```
Copy RAM:Current_Work to Modules:Libs/Design clone
```

or

```
XCopy RAM:Current_Work to Modules:Libs/Design
```

Either above command copies all files in the RAM:Current_Work directory to the Modules:Libs/Design directory preserving all date/time stamps.

Windows In Intuition

The most important aspect of any programming environment is how it interacts with the display screen. You can't create much of a program without showing something on the screen. In the case of the Amiga this is especially important since its graphic processing capability is far superior to any other computer in its class. Many people purchase the Amiga specifically to write programs that cause things to occur on its display screen. To meet that need Intuition provides a wide range of functions, each involving a rather large number of details. The key to dealing with the complexity of all this is to break each Intuition function down into its most elementary operational steps. By doing so you will discover that many of them work according to the same general operational theme, thus facilitating the learning of all of them. Here is the theme used by most functions that create graphic objects:

1. Declare a structure in which to store a description of the graphic object you wish to create. This will not be the graphic object itself, but only a description of it.

2. Declare a pointer that will eventually hold the address of the graphic object you wish to create. Note that this is not an instance of the graphic object, only a pointer to one. It is usually initialized to NULL.

3. Open "intuition.library" if you have not done so already.

4. Assign to each member of the description structure whatever characteristics you want your graphic object to have. This is usually done with a series of assignment statements.

5. Call the Intuition function that creates the object using as an argument the address of the description structure declared in step one and assign the value returned by the function to the pointer declared in step two. Note that it is a common theme for Intuition functions to return the addresses of the graphic objects that they create. Often a cast (K&R, page 45) is required when assigning the address returned by a function.

6. Test the value returned by the function to determine if the object was properly created. Usually you will want to see if it is a legal address or NULL. Null signifies that the graphic object was not created and you must take corrective action.

7. Use the pointer to the object (as well as other pointers derived from it) throughout your program to access its features or to control it in different ways.

8. Close the object before your program terminates, removing it from the screen and returning memory back to the system.

This eight point operational theme will take you a long way in your study of intuition. Of course what you actually do with a graphic object may vary greatly from program to program, but the above theme often forms a background plan around which your program works. The Intuition function that is used to open a window follows the above theme very closely.

Figure One
Sugested File Structure for Archiving Module

Archive Diskette

```
Modules: ──── Libs ──── Design ──── Libs.c─┐
                          │                 ├─ Libs.h
                          │                 ├─ Libs.o
                          │                 └─ LMKFile
                          │
                          ├─Test───── Libs.h
                          │           Libs.o
                          │          ─Test_Libs
                          │          ─Test_Libs.c
                          ├─Libs.o    └─LMKFile
                          │
                          └─ Libs.h
```
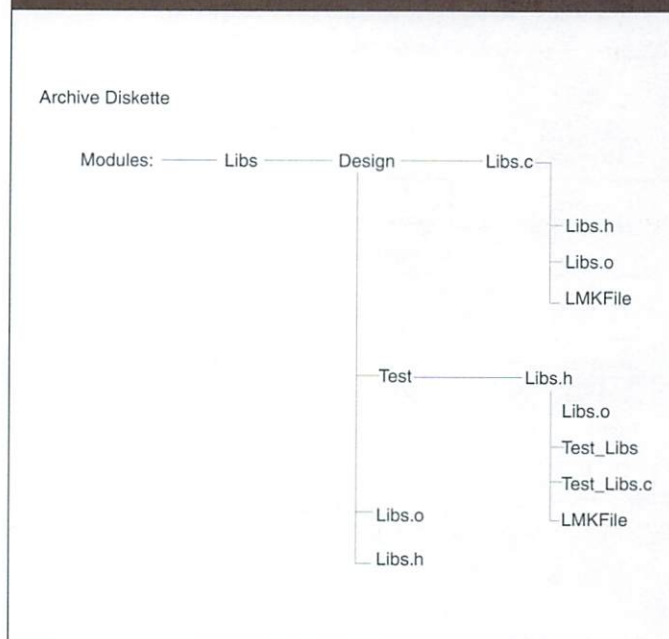
The OpenWindow( ) Function

Your knowledge of the Amiga so far should be enough that I don't need to tell you that a window is a rectangular area on the screen that is used by your programs to display the results of various text and graphic processing. But you should know a bit more than that before attempting to open a window in C. Windows can be of different sizes and located at different locations within their parent screen. They have the ability to display colors using a concept called "color indirection", in which a large number of possible colors are accessed indirectly through a smaller number of hardware pen registers. The total number of different colors that they can display simultaneously depends on a property of the window's parent screen called depth. Windows can also have attached to them gadgets that allow them to be positioned and re-sized by the user independent of your program.

This long list of characteristics and much more can and should be studied initially in BASIC rather than C. If you are unfamiliar with any of them I recommend that you do that. Naturally I will demonstrate how each is implemented in C, but I cannot spend time here explaining the features themselves. I recommend the text "Advanced AmigaBASIC", by Halfhill and Brannon, Compute Publications, 1986.

```
MyWindow = (struct Window *)OpenWindow(struct NewWindow *Window_Description)

     INPUT                                    OUTPUT

    address of                           Address of Window
   description                           created in memory
    structure
    _____                             _____
   | & operator |        _____      | Window Structure |
   | or pointer |       |          |     |                  |
   |            |\      | OpenWindow |    |   created by     |
   |            | \_____|          |_____|   OpenWindow()    |
   |_____| /     | function  |    |                  |
                                          |_____|
                                                  /|\
       \|/                                         |
    _____                                    |
   | NewWindow |          /\  No (address = NULL)   |
   |Description|         /  \                        |
   | structure |        /valid_____   exit      |
   |           |        \  ?  /          \  program  |
   | TopEdge = |         \   /              safely   |
   | Width   = |          \ /                        |
   |  ...    = |           |                         |
   |_____|     YES (address valid)             |
                           |                         |
                    Assign MyWindow pointer          |
                    to Window structure              |
         \|/          created above                  |
        _____                                   
       | MyWindow |_____
       |          |
       |_____|

           \|/

      continue with normal
      program execution
```

Within the context of Intuition a window is a graphic object created by the OpenWindow() function. This function accepts as an argument the address of a structure that contains a description of the window you want, and it returns the address of the window that it creates for you, or a NULL if for some reason it cannot fulfill your request. Your program will interact with this window object primarily by knowing its address. Naturally C, being a high level language, provides the concept of pointers (K&R, page 93) to deal with such issues and as a result operations involving windows rely heavily on the use of various pointers. But lets start at the beginning.

Here are the operational steps required to create and use a window:

1. Declare and initialize a structure of type "NewWindow" to hold a description of the window you want to create. The definition (template) of the NewWindow structure appears in the header file <intuition/intuition.h>.

2. Declare a pointer to "struct Window" to hold the address of the window that will be created by the OpenWindow() function. Note that this not an instance of a Window structure, but only a pointer to one. Initialize this pointer to NULL. The definition (template) of the Window structure appears in the header file <intuition/intuition.h>.

3. Open "intuition.library" if you have not already done so.

4. Using a series of assignment statements store in each member of the NewWindow structure the characteristics of the window that you want.

5. Call the OpenWindow() function using as an argument the address of the description structure and assign the value returned by the function (address of the window created) to the pointer declared in step two. Cast the assignment to the correct type.

6. Test the value that was returned by the OpenWindow() function. It will be either a legal address for a properly created window, or a NULL if for some reason one could not be created. Take corrective action if it is NULL. Any attempt to use a window through a NULL pointer will crash your machine.

7. Use the pointer to the Window (as well as other pointers derived from it) throughout your program as an argument in various functions that display text and graphic images.

8. Close the Window before your program terminates, removing it from the screen and returning memory to the system.

See Figure Two for a graphic representation of the operation of opening a window.

NewWindow Structure

This is the structure in which you store all the characteristics of the window you want the OpenWindow() function to create for you. I call it the description structure. Like all structures it consists of a list of members of different types (K&R, page 127). View it as a kind of shopping list. Below I list its members as defined in <intuition/intuition.h> header file, with the exception that I have removed all the comments.

```
struct NewWindow
{
    SHORT LeftEdge;
    SHORT TopEdge;
    SHORT Width;
    SHORT Height;
    UBYTE DetailPen;
    UBYTE BlockPen;
    ULONG IDCMPFlags;
    ULONG Flags;
    struct Gadget *FirstGadget;
    struct Image *CheckMark;
    UBYTE *Title;
    struct Screen *Screen;
    struct BitMap *BitMap;
    SHORT MinWidth;
    SHORT MinHeight;
    USHORT MaxWidth;
    USHORT MaxHeight;
    USHORT Type;
};
```

Some of them are self-explanatory, while others seem a little obscure. Let's go through them all quickly, just to get a general idea of what they do.

The LeftEdge and TopEdge members of this structure refer to the position of those edges of the window from the top left corner of its parent screen, measured in pixels. Width and Height are its size, also measured in pixels. The DetailPen member refers to the hardware color register used for any text that you want to appear in the window's title bar. BlockPen is the register used for the background of the title bar and borders (if any). Most windows use register 0 (blue) for DetailPen and 1 (white) for BlockPen, but you may use whatever registers you like. The IDCMPFlags member refers to a special message system that is used to communicate with the window via the keyboard or mouse. The Flags member refers to several features, such as whether the window will have system gadgets, whether it will be active when it opens, or how it will deal with remembering its contents after being resized or temporarily covered up by other windows. Some of these features affect the previous IDCMP message system as well. The Gadgets and CheckMark members are used if you want the window to have gadgets or a menu checkmark of your own design. The Title member points to a string constant that you want to appear in the window's title bar. The Screen member points to the screen in which you want your window to be placed. The BitMap member points to an area of display memory required if you want to use a concept called SUPER_BITMAP, which is another approach to having your window remember its contents after being re-sized by the user. MinWidth through MaxHeight refer to the dimension extremes beyond which you do not want the user to adjust the size of the window. And finally, type specifies whether you want your window to appear in the Workbench screen or in one that you create yourself - more about that next issue.

I understand that the above overview of the NewWindow structure was very superficial. Don't worry, I will come back to it next issue. It is more important now that you learn the general operation of the OpenWindow() function and not get bogged down by details. Besides, as the following example will demonstrate, those members that control unfamiliar, advanced features can simply be set to NULL.

First Window Example

Create a project drawer and copy into it the First_Window.c and its corresponding LMKFile file from the magazine diskette. Then copy into that same drawer the Libs module (Libs.o and Libs.h). Do you understand why Libs.c is not needed?

```
FIRST_WINDOW.C

/*                      First_Window.c                    */
/*                                                        */
/*  AC TECH                            Volume 1, Number 3 */

#include <intuition/intuition.h>
#include <proto/intuition.h>
```

```
#include <proto/graphics.h>
#include <proto/dos.h>
#include <stdio.h>
#include <stdlib.h>

#include "Libs.h"

VOID main(int argc, char *argv[]);


VOID main(int argc, char *argv[])
{

    /* === Step 1: Declare description structure === */

    struct NewWindow Window_Description;


    /* === Step 2: Declare and initialize Window pointer === */

    struct Window *MyWindow = NULL;



    /* === Step 3: Open Libraries to use Intuition === */

    if(!Open_Libs())
    {
        printf("Trouble opening libraries\n");
        Delay(100);
        exit(RETURN_WARN);
    }


    /* === Step 4: Assign description structure members. === */

    Window_Description.LeftEdge   = 100;
    Window_Description.TopEdge    = 50;
    Window_Description.Width      = 400;
    Window_Description.Height     = 100;
    Window_Description.DetailPen  = 3;
    Window_Description.BlockPen   = 2;
    Window_Description.Title      = "My First Window";
    Window_Description.Flags      = ACTIVATE | SMART_REFRESH;

    Window_Description.Screen     = NULL;
    Window_Description.Type       = WBENCHSCREEN;

    Window_Description.IDCMPFlags  = NULL;
    Window_Description.FirstGadget = NULL;
    Window_Description.CheckMark   = NULL;
    Window_Description.BitMap      = NULL;
    Window_Description.MinWidth    = 0;
    Window_Description.MinHeight   = 0;
    Window_Description.MaxWidth    = 0;
    Window_Description.MaxHeight   = 0;

    /* === Step 5: Call OpenWindow() function and assign return === */
    /* ===          address to Window pointer.                 === */
    /* ===                                                     === */
    /* === Step 6: Test value returned by OpenWindow.          === */

    if(!(MyWindow=(struct Window *)OpenWindow(&Window_Description)))
    {
        printf("Trouble opening MyWindow\n");
        Delay(100);
        Close_Libs();
        exit(RETURN_WARN);
    }

    /* === Step 7: Use window === */

    SetAPen(MyWindow->RPort, 1);
    Move(MyWindow->RPort, 93, 50);
    Text(MyWindow->RPort, "Help me, I'm stuck in here.\n", 27);

    Delay(500);
```

```
    /* === Step 8: Close window === */

    if(MyWindow)
        CloseWindow(MyWindow);

    Close_Libs();

    exit(RETURN_OK);
}


LMKFILE:

PROGRAM = First_Window
LCFLAGS = -b1 -cfistc

$(PROGRAM): $(PROGRAM).o Libs.o
    BLink FROM LIB:c.o+$(PROGRAM).o+Libs.o TO $(PROGRAM) LIBRARY LIB:lc.lib

$(PROGRAM).o: $(PROGRAM).c Libs.h
    LC $(LCFLAGS) $(PROGRAM)
```

Let's start with the subject at hand, the opening of the window. Step 1 creates an instance of a NewWindow structure in which we will store a description of the window we want. I named it Window_Description. You are free to use any name you want. Step 2 declares a pointer of type "struct Window", names it MyWindow (you can use any name) and initializes it to NULL. This is the pointer to which you will assign the address of the window when it is returned by the OpenWindow() function. Step 3 opens "intuition.library". Step 4 stores a description of the window into the description structure, using a series of assignment statements. I have arranged them in the listing into related groups.

Of particular interest are the NULLs at IDCMPFlags, FirstGadget, CheckMark, and BitMap, which mean that I do not want to use those features. Notice the NULL at the Screen member and "WBENCHSCREEN" at the Type member. Together these two assignments cause the window to open up on the Workbench screen. The macro "WBENCHSCREEN" is defined in the header file <intuition/screens.h>. The MinWidth through MaxHeight members are all set to zero because the window I want to create will not have any re-sizing capability. Now look at the DetailPen and BlockPen members. The values I have chosen should demonstrate beyond any doubt what each one does. The result will be an orange title (color 3 in DetailPen) in a black border (color 2 in BlockPen).

The last member I want to mention is Flags. I have assigned it the value "ACTIVATE". This is a macro defined in the header file <intuition/intuition.h>. As you know, windows can be activated or de-activated using the mouse. De-activated windows have a hazed or ghosted appearance in their title bar. I asked for the window to be automatically activated to improve the appearance of its title bar. Step 5 and 6 of the general operational theme are accomplished together in one tight "if block". The window pointer is assigned, logically inverted, and finally tested for success all in one line using the popular functional calling style that I presented last issue.

In the event that a window cannot be created, the NULL returned by the OpenWindow() function gets inverted to 1 (TRUE) causing the if() statement to succeed and the corrective action inside the block to execute, safely terminating the program. Otherwise, the window address is considered valid and the

program pushes onward. Step 7 involves actually using the window. To do that you usually have to specify the window pointer, or an address derived from it, as an argument in a graphic function. I demonstrate this with three lines that together position and display a message inside the window. Never mind exactly how these functions work, you will learn that soon enough. For now simply view them as the part of the program that accomplishes step 7 of the operational theme. Finally, and with no surprise, comes the last step which closes the window. To do that you must call the Intuition function CloseWindow() using as an argument the pointer to the window that was assigned in step 5. Notice the similarity to the closing of libraries last issue. Although it is not necessary in this program it is best to test the pointer before closing the window, otherwise the computer could crash.

Initializing Structures

There is another way in Standard C to initialize members of a structure. The method consists of initializing the entire structure (all members) automatically when it is declared, like this:

```
struct NewWindow Window_Description = {100, 50, 400, 100, 3, 2,
                    NULL, ACTIVATE, NULL, NULL,
                    "My First Window", NULL, NULL,
                    0, 0, 0, 0, WBENCHSCREEN};
```

The above instruction causes the compiler to initialize each member of the NewWindow structure consecutively using the values appearing in the list. This method is preferred by some programmers over a long series of assignment statements because it is both faster to type in and faster to execute. But there are problems with this method. If one number in your data is out of place then all members following it may end up receiving wrong values. But more importantly, this quick method does not use the structure's descriptive member names. You need those names in order to understand the exact purpose of each piece of data in the program.

If you want to verify the purpose of one number in the above list you will have to search through the header file system for the structure's template and then count down its list of member names until you get to the one you want. Chances are you will make a mistake. In contrast initializing that same structure using individual assignment statements puts those member names right within the block of code where they are used, where they can help both you and others understand the program's operation. In addition, individual assignment statements can be arranged in any order. Grouping related ones together, as I did earlier, helps to better understand their collective operation.

One more reason for not using the above quick initialization method for a description structure is that you will almost certainly want to write a pre-compiled module that contains a function capable of opening windows having a range of characteristics. This is in accordance with the philosophy of structured programming and information hiding, and we will be doing that next issue. You will see then that part of the design of such a function is the interpretation of arguments passed to it and the assignment of values to corresponding members of the description structure. In such situations the above quick initialization method will be of little value.

LMKfile Macro Substitutions

The LMKFile for this last example uses a new concept, macro substitutions. Two macros are defined at the top of the file for substitution at different locations within it. The first of these is the line:

```
PROGRAM = First_Window
```

This line defines the word "PROGRAM" to be equivalent to the text string "First_Window". The dollar sign operator with parentheses causes the macro to be substituted at different locations within the file. Thus the instruction:

```
$(PROGRAM): $(PROGRAM).o Libs.o
    BLink FROM LIB:c.o+$(PROGRAM).o+Libs.o TO $(PROGRAM) LIBRARY LIB:lc.lib
```

is equivalent to:

```
First_Window : First_Window.o Libs.o
    BLink FROM LIB:c.o+First_Window.o+Libs.o TO First_Window LIBRARY LIB:lc.lib
```

The target file First_Window is produced by linking First_Window.o, Libs.o and the system startup code LIB:c.o.

Similarly the second macro:

```
LCFLAGS = -b1 -cfistc
```

gets substituted at the proper place as an option field for the LC command.

Macro substitutions allow you to use the same LMKFile for several projects. You just change the macro definition of PROGRAM and it will compile whatever new project you want, assuming of course that you want it to be compiled and linked in exactly the same manner.

Example Programs on Disk

Modular programming is more practice than theory. To help you along with that I have included on disk two example projects that you can compile yourself. They are called Ball_Anim and Line_Anim and each represent an example that I will be presenting next issue. They will help you gain valuable experience compiling projects consisting of pre-compiled modules. Note that you will have to copy the Libs module (the Libs.o and Libs.h files) into each project drawer. I'm expecting you to do that yourself. Once you have done that you can go ahead and double-click the "Build" icon, or enter LMK into a SHELL window. Observant readers will notice the presence of a second module called Display.o, installed in each project drawer. Don't worry that you do not yet understand the purpose of this module. I will be presenting it next issue. Your main goal here is to go through the motions of compiling the project, and that won't change whether you understand the exact purpose of the modules or not. Notice also that the source code for Display.o is not present. It isn't necessary. Like Libs.o it is a pre-compiled module that gets added to your program by the BLink program. The only programs that actually get compiled are Ball_Anim.c, Line_Anim.c, and the various header files.

→

I have entered the proper instructions into the LMKFile for each project. Make sure you look at it and try to understand its instructions.

For those readers who do not own SAS/C I provide compiled versions of both projects in the drawer called "Compiled_Programs". Simply double-click their icons. Perhaps after seeing them you will want to learn more about how they work, and even try your own hand at writing similar programs yourself. An Amiga with a C compiler is an exciting combination. Note however that these programs are not intended to be competitive animations in their own right. Rather, they are instructional examples that will be used next issue to help me present some fundamental programming concepts.

More On Disk

Before leaving this article remember that the diskette that comes with this magazine contains some valuable background material to help you better understand the configuration programs given in the last issue of AC's TECH/AMIGA. You will also find that material helpful if you want to make modifications to those installations yourself.

Next Issue!

Today I barely introduced the concept of opening windows in Intuition. Next issue I will present some of its more intricate details, as well as a streamlined version in the form of a pre-compiled module, called Display.o (you knew that was coming didn't you). I will include in its design the ability to display different resolutions and number of colors. In addition I will present a subject that is almost always ignored in programming articles but which is very important, screen scaling. If you have ever programmed in AmigaBASIC you probably know that the dimensions and position of graphic images are measured using pixel numbers, measured from the top left corner of a window. Many of the Amiga's graphic commands accept these pixel numbers directly. Unfortunately most graphic images depend on some sort of mathematical equation which cannot be expressed directly in terms of pixel numbers. I have therefore designed into the Display.o module two scaling functions that will allow you to express equations in their conventional form, while at the same time pass to the Amiga's graphic functions their corresponding pixel numbers. In fact, many of our programs will be written in such a way that they work regardless of the actual number of pixels in a window. Hard to believe? Try the following short experiment in the Line_Anim.c program. Modify line 46 from:

```
if(!Open_My_Screen("LACEHIGH4", &my_screen, &my_svp))
```

to

```
if(!Open_My_Screen("LOW4", &my_screen, &my_svp))
```

That is, change the resolution from LACEHIGH4 to LOW4 (high resolution interlace to low resolution non-interlace, more about this next issue). Compile the program again and execute it. Notice that even though the text characters are now a different size (larger) they still appear properly centered in the window. Also, the animation itself remains well behaved, preserving its original physical size despite of the fact that the number of pixels on the screen, both horizontally and vertically, is now only half of what it originally was. This kind of resolution independence does not happen automatically but is a result of the way we write our programs. I hope to see you next issue.

☑

Recently, we were ordered by U.S. military officials to explain to their complete satisfaction just what a SuperSub is (as we all know, it's the best subscription deal around for Amiga users, since it includes both *Amazing Computing* and *AC's GUIDE*).

★★★★★

Then, a prominent Congressman wired to ask us if we would testify before a top-secret subcommittee as to whether or not we can produce a single prototype SuperSub for less than $500 million (is this guy kidding? – a one-year SuperSub costs just $36 – and we can produce one for *anybody*!).

★★★★★

Finally, a gentleman called us from Kennebunkport and told us to read his lips, but we told him we couldn't, because we don't have a picturephone.

And then he ordered a SuperSub.

AC's SuperSub –
It's Right For You!
call 1-800-345-3360

List of Advertisers

Please use a Reader service card to contact those advertisers who have sparked your interest. Advertisers want to hear from you. This is the best way they have of determining the Amiga community's interests and needs. Take a moment now to contact the companies with products you want to lean more about. And, if you decide to contact a advertiser directly, please tell them you saw them in

AC's TECH AMIGA

An Introduction to Programming for HAM-E

by Ben Williams, Black Belt Systems

About the HAM-E

The HAM-E is an external "display enhancer" for all Amiga models, NTSC and PAL. The term "HAM-E" stands for "Hold-And-Modify-Expander". The HAM-E actually provides two new display modes. One is 256 simultaneous colors from a 24 bit palette. The other is 262144 simultaneous colors using an 8 bit per pixel extended HAM mode. The extended HAM mode has been designed such that it is much less prone to "fringing" than the HAM mode that is standard with the Amiga. The 256 color mode isn't HAM, and as such has no drawbacks whatsoever.

The HAM-E provides the Amiga with two new modes, and passes through the graphics modes that are "native" to the Amiga. The two new modes, which we call HAME (no dash) and REG mode, appear on standard Amiga screens that pull up and down, go front to back, and overlay or underlay other screens in a 100% compatible manner.

Developers wishing to make their applications compatible with the HAM-E receive the utmost in support from Black Belt's technical personnel. Most applications can be supported with a minimum of effort on the part of the developer. Applications that run entirely on the HAM-E will of course require more effort than simple output of images (such as a ray tracer might want to do), yet the new display modes are easy to use, and we can help you all through the process.

What's the point?

The objective of this article is threefold. First, we want you to know that you can work with the HAM-E display system easily. Next, we'd like to give you an overview of what methods are available, and how we might get involved with you, if you require assistance. Finally, we'd like to help you to understand how the HAM-E works. We'd like you to understand the HAM-E device to the extent that you may effectively consider programming for it in relation to projects you may be involved with.

Programming approaches

Programming for the HAM-E device can be approached at three different levels. You can use our libraries, which are freely available to developers. You can use our C language source code as a basis for your own code. Finally, you can program to our specifications, which are clearly set out for you.

The Library Interface

One of the quickest ways to provide HAM-E support in your application is to use one or both of the libraries we provide. There is a low-level library called hame.library, and a higher-level library called renderhame.library.

The low level library provides a great deal of functionality, at very little cost in memory. It is a standard Amiga library, you simply open it and then you can call the routines contained there.

Library Interface, continued...

The routines called out in the hame.library function table are extremely versatile, and can form the basis for almost any imaginable low level manipulation of a HAME or REG mode screen.

The facilities are here for building a game from scratch, complete with "BOB- like" routines that allow you to use the blitter (with complete masking) to move game objects around the screen. Color cycling of any number of registers allows a number of animation techniques without any actual data movement on the display. We've even provided a fast routine to move 8-bit buffers in fast memory into HAM-E mode screens in large chunks, as efficiently as possible so that you can handle your objects in fast ram, and then use the library to move the object to the screen quickly. Once there, you can clip it with the blitter routines provided, and you have a complete BOB.

If your desire is to read HAM-E mode screens, that's all here too. all you have to do is use the GetRGBPixel() routine for each pixel in the image, and you'll have a full 24-bit readout of the image.

On the other hand, if you are looking for output in one of the HAM-E modes, for example you might be writing support code for a ray-tracer, or some other application that needs to take it's internal RGB buffers and turn them into a HAME mode image, the renderhame.library is the tool you would use. This requires that you also open the hame.library, however you won't be using any of the low level tools—the renderhame.library uses these tools, which is why you need to open the hame.library for this type of operation.

Table of Library functions for "hame.library"

HAME_Init()	tells the library mode to use, basic setup function
HAME_Dispose()	Frees the structure allocated by HAME_Init()
HAME_SetAPen()	Pick primary drawing pen (0-255)
HAME_ReadPixel()	returns 8 bit value of pixel at x,y
HAME_WritePixel()	writes 8 bit value to x,y
HAME_GetRGBPixel()	returns 24 bit RGB value for pixel at x,y
HAME_GetRGB8()	returns 24 bit color register setting
HAME_SetRGB8()	writes 24 bit value to color register
HAME_Ellipse()	Draws an ellipse, filled or unfilled in "APen"
HAME_Line()	Draws a line in "APen"
HAME_Box()	Draws a box, filled or unfilled in "APen"
HAME_OpenScreen()	Aid for BASIC language users
HAME_CloseScreen()	complement to HAME_OpenScreen()
HAME_CycleLeft()	cycles the color registers
HAME_CycleRight()	cycles the color registers
HAME_Move()	moves the current pen position; no drawing
HAME_Draw()	draws from the current position to x,y
HAME_8BitRender()	moves an 8 bit "byte" buffer into the HAME screen
HAME_Scroll()	scrolls the HAME screen
HAME_OpenFont()	opens a standard Amiga font or a colorfont
HAME_GetFontPalette()	stuffs colorfont palette into the HAME palette
HAME_SetFontPen()	for both colorfonts and regular fonts
HAME_Text()	actually writes the text to the screen
HAME_MakeFTables()	setup for HAME_QText()
HAME_QText()	very fast font rendering, more memory overhead
HAME_CloseFont()	complement to HAME_OpenFont()
HAME_AllocClip()	creates blank clip for your use
HAME_GetClip()	blits clip from screen region
HAME_DisposeClip()	complement to HAME_AllocClip(), HAME_GetClip()
HAME_PutClip()	blits clip to screen (optional masking)
HAME_ReadClipPixel()	returns 8 bit value at x,y in clip
HAME_WriteClipPixel()	places 8 bit value at x,y in clip
HAME_ReadMaskPixel()	returns 1 bit value at x,y in clip mask
HAME_WriteMaskPixel()	places 1 bit value ay x,y in clip mask
HAME_AllocClipMask()	creates a mask for a particular clip
HAME_DisposeClipMask()	complement for AllocClipMask()
HAME_MakeClipMask()	creates mask using transparency supplied
HAME_WaitScanLine()	scan line delay to avoid flicker in games
HAME_LockLayer()	Prevents system operations from trashing display
HAME_UnLockLayer()	Re-allows system to render to the HAME screen

Table of Library functions for "renderhame.library"

HAME_RenderHam()	creates 18 or 24 bit output from RGB buffers
HAME_RenderReg()	creates 256 color output from RGB buffers
HAME_SaveIFF()	saves resulting render as viewable IFF file

The C Code approach

Black Belt has made available a great deal of C language source code which can be used as a model for many types of HAM-E operations. For instance, the entire source code to version 1.0 of our register mode paint program is on our BBS (at 406-367-ABBS/2227). In there, you can find routines to do almost anything, from pixel level access to blitter handling.

C coding for the HAM-E is relatively straightforward. We suggest that you use our "canned" low-level routines as much as possible, to save yourself a great deal of coding effort. Or, if you feel that you could write higher performance code, you can use them as examples. In either case, the path to support via custom C code is well covered.

Programming "to the specification"

The HAM-E device is fairly easy to write software for, in any language that allows you fast access to the Amiga's chip memory.

The HAM-E uses "pairs" of four-bit, high resolution pixels to encode the output pixels. The Amiga maintains a high-resolution, four bitplane display. The HAM-E device takes pairs of these pixels at four bits each, and essentially turns them into single pixels that are eight bits each. These pixels are then used to load color registers, select pixel values, and so on. The Amiga doesn't know what's going on "behind it's back" so to speak, and so handles the screen as it normally would.

REG Mode

For REG mode, these 8 bit combination pixels are simply used to create an 8- bit pixel, which chooses a color register number from zero to 255.

HAME Mode

For HAME mode, the 8 bit combination pixels are used to specify a hold-and- modify command to the display adaptor. We use the top two bits in the 8 bit word as 'control' bits, and the bottom six bits as data bits. If the top two bits are not zero, then the bottom six data bits are loaded into either the red, green or blue gun output, while the other two of the three output colors are 'held' at the value provided by the previous pixel.

If the two most significant bits are zero, then the bottom six bits are used to specify one of sixty color registers, or, to switch to one of three other banks of sixty color registers. If a new color register is chosen, then all three of the red, green and blue guns change to the 24-bit value that is in that color register, providing a "sharp edge" color change. As mentioned before, there are 60 available. If a new bank is chosen, then all three of the red, green and blue guns are held through the pixel that causes the bank switch. This allows the image to contain up to 240, 24-bit color registers which can be used to change colors quite sharply, without fringing.

Interlace Operation

When the Amiga operates in Interlace mode, it's not really sending out a 400 line image. Instead, it's sending two completely separate 200 line images, which are simply placed on the monitor screen so that they "interleave", or interlace, with each other.

The HAM-E hardware is not aware of this. It treats each of the 200 line images as completely separate objects, each with it's own set of up to 256 (or 240, in the HAME mode) color registers.

As a result, you can actually have up to 512 color registers in a REG mode image, and up to 480 color register in a HAME mode image. These are divided into two sets of registers, one for the odd lines in the image, and one for the even lines in the image.

For normal images, this really doesn't turn out to be a large advantage, although it can help somewhat. But there is a special class of images where this is really, really nice to have. That is, the 3-d images used by the X- Specs glasses, manufactured by Haitex.

These glasses operate by using a Liquid Crystal Display technology to create a "shutter" over the entire lens on each eyepiece of the glasses. Then, the Amiga is made to present the image for one eye in the even field of the interlace image, while the left eye is shuttered, or blocked off. Next, the image for the other eye is presented in the other interlace field, while the right eye is shuttered. The result is that a stereo pair of images is presented to the user, and the HAM-E (remember the HAM-E?) can render these images with completely independent color registers, and therefore complete independence between the subject matter of the images.

Conclusion

Amiga developers who wish to join the those who are already supporting the HAM-E device (ASDG, Holosoft, Progressive Peripherals, Virtual Reality Laboratories, Oxxi, and many, many more) have many possible levels of approach available to them. The HAM-E provides a great deal of functionality at a reasonable cost to the end user, and a large number are in user's hands.

If you have thoughts for library functionality that we've overlooked, please contact us and we'll be extremely pleased to listen to your ideas. If your ideas seem reasonable, we'll put them into the libraries as quickly as possible.

Black Belt Systems technical support is always available during mountain time business hours to provide additional support, and we consider helping developers with their efforts in this matter our top technical priority. If you have questions about the HAM-E and developing software for it, please call our tech support line at (406) 367-5509 and ask for Barry or Ben. We'll be glad to talk to you!

✓

Would like like to see in-depth articles about programming specifica pieces of equipment like the HAM-E, in AC's TECH?

Let Us Know!

Write to:

AC's TECH Suggestions
P.O. Box 869
FAll River, MA 02722-0869

Using RawDoFmt in Assembly

If you have a need to print formatted strings in assembler, stop wasting time and code rolling your own routines and read on.

by Jeff Lavin

Although the Exec library is rich in programmer-accessible functions, there is one extremely useful function that is very much under-represented in assembly language programs, particularly those written by beginners. This is probably because the documentation is confusing. If you have a need to print formatted strings, stop wasting time and code rolling your own routines and read on.

RawDoFmt(), pronounced "raw do format", is used in a number of Exec's higher functions including KPrintf(), found in the debug.lib, DPrintf(), found in the ddebug.lib, and of course ROMWack, the system debugger. It is capable of handling most standard 'C' type formatting with the exception of floating point. RawDoFmt() packs an amazing amount of functionality into less than 600 bytes. All of the following 'C' type formatting is supported:

```
%d - print an integer
%c - print a character
%x - print an unsigned hexadecimal integer
%s - print a string
```

Additionally, the following modifiers are supported:

```
Justification:    %-d  - print an integer left-justified
Lead char select: %08d - fill leading space w/zeros instead of blanks
Field width:      %8d  - print an integer in a field 8 spaces wide.
Field truncate:   %8.4d - only print 1st 4 chars of integer
                          in a field 8 spaces wide.
Data size:        %ld  - print a long integer.
```

Integers are used here as examples, but these modifiers work with other data types as well. With the exception of the modifiers for size and for leading zeros/blanks, they also work with strings.

The register usage for RawDoFmt() is:

```
RawDoFmt (FormatStr, DataStream, PutChProc, PutChData)
          A0         A1          A2         A3\
```

where:

FormatStr = Pointer to 'C' language type null-terminated format string.

DataStream = A stream of data that is interpreted according to the format string. Defaults to WORD size. BYTE size data is *not* supported. Pointers to strings are always LONG. Also note that the data *must* match the sizes called for in the FormatStr.

```
PutChProc = The user-supplied procedure called with each char output.
```

For example:

```
PutChProc (Char, PutChData)
           DO:8  A3
```

PutChData = An address register passed through to PutChProc(). This is generally used as a pointer to a character buffer, which *must* be big enough to hold any possible formatted string.

If you find this confusing, you're in good company! Basically, to use RawDoFmt(), you need to do the following:

```
        lea     (DataStream,pc),a1      ;Ptr to Data
        movea.l a1,a2
        move.w  d0,(a2)+                ;Data: An Int
        move.l  d1,(a2)+                ;Data: A Long
        move.w  #19020,(a2)+            ;Data: Another Int
        lea     (SomeString,pc),a0      ;Data: A ptr to a string
        move.l  a0,(a2)
        lea     (fmtString,pc),a0       ;Ptr to FormatStr
        lea     (PutChProc,pc),a2       ;Routine to do whatever you want
        lea     (CharBuf,pc),a3         ;Buffer for use by PutChProc
        movea.l (AbsExecBase).w,a6      ;Exec library base
        jsr     (_LVORawDoFmt,a6)       ;Call the routine

fmtString db    'Int: %d, Long: %ld, HexInt: $%d, String: %s.',0
```

The data could be passed on the stack instead, like so:

```
        pea     (SomeString,pc)         ;Push a ptr to a string
        move.w  #19020,-(sp)            ;Push another Int
        move.l  d1,-(sp)                ;Push a Long
        move.w  d0,-(sp)                ;Push an Int
        movea.l sp,a1                   ;Get DataStream ptr into proper
                                        ;register
```

Remember that data pushed on the stack is accessed in reverse order!

Then, when you return from RawDoFmt():

```
        lea     ($C,sp),sp              ;Fix stack before returning
```

There are a number of things you can do with the PutChProc() function. In this example we are writing each character to stdout as we get it:

```
PutChProc movem.l d2-d3/a6,-(sp)        ;Save stuff
        move.b  d0,(a3)                 ;Put char in buffer
        move.l  (_stdout),d1            ;Filehandle previously opened
        move.l  a3,d2                   ;Buffer
        moveq   #1,d3                   ;Length
        movea.l (_DosBase),a6           ;Dos library base
        jsr     (_LVOWrite,a6)          ;Write the char to stdout
        movem.l (sp)+,d2-d3/a6
        rts
```

The problem with this example is that single character I/O takes a tremendous amount of overhead for each character. Here's a better way:

```
PutChProc move.b  (a3)+
        rts
```

When you return from RawDoFmt(), find the length of the buffer and write it all out at once. Output from RawDoFmt() is always null terminated.

Several more points, and on to the real code. D7 is the only register besides A3 that RawDoFmt() doesn't save, *and* also passes on to PutChProc(). If you want a count of the characters PutChProc() puts into your buffer, without doing a strlen() when it returns, you need to do the following:

```
MyFormat movem.l d7/a2-a3/a6,-(sp)      ;Save stuff
        lea     (DataStream,pc),a1      ;Ptr to Data
        lea     (SomeString,pc),a0      ;A ptr to a string
        move.l  a0,(a1)
        lea     (fmtString,pc),a0       ;Ptr to FormatStr
        lea     (PutChProc,pc),a2       ;Char fill procedure
        lea     (CharBuf,pc),a3         ;Buffer for use by PutChProc
        movea.l (AbsExecBase).w,a6      ;Exec library base
        moveq   #0,d7                   ;Clear character count
        jsr     (_LVORawDoFmt,a6)       ;Call the routine
        move.l  d7,d0
        subq.l  #1,d0                   ;Count of characters, excluding null
        movem.l (sp)+,d7/a2-a3/a6
        rts

PutChProc move.b  (a3)+                 ;Put character in buffer
        addq.l  #1,d7                   ;Add 1 to character count
        rts

fmtString   db    'Mary had a little lamb %s',0
SomeString  db    'whose fleece was white as snow.',0
```

When this subroutine returns, the character count will be in D0. If there were no characters output, D0 will contain -1. You can test both of these by checking the flags:

```
        bsr     MyFormat
        bpl.b   GotSomeChars            ;Branch on 0 or greater
        ...                             ;Got no characters
```

As stated before, just as with C's printf(), you *cannot* pass bytes to RawDoFmt(). In order to print a byte size or character data, you must pass it as a word:

```
        moveq   #0,d0
        move.b  #27,d0
        move.w  d0,(a1)                 ;A1 is pointing to DataStream
  or
        clr.w   (a1)                    ;A1 is pointing to DataStream
        move.b  #'a',(1,a1)
```

In addition to not being able to pass characters as bytes, you can't pass the percent sign (%) in a format string at all! You can't use '%%' here to get a single %, the way you can in printf(). So how do you print a single percent sign? Easy:

```
        move.w  #87,(a1)                ;We fervently hope & pray!
        moveq   #0,d0
        move.b  #'%',d0
        move.w  d0,(2,a1)               ;A1 is pointing to DataStream

fmtString   db    'AC's Tech readers who liked this article is %d%c.',0
```

Finally, because RawDoFmt() is only capable of printing unsigned integers, you might think there would be a problem if you needed to print signed numbers. Here's one way:

```
SignedWord  lea     (Pos.fmt,pc),a0        ;Default to positive
            tst.w   d0                     ;Check our WORD size data
            bpl.b   .notneg                ;Branch if positive
            lea     (Neg.fmt,pc),a0        ;OK, get negative format string
            neg.w   d0                     ;Make number positive
.notneg     lea     (_FmtArgs-DT,a5),a1    ;Ptr to DataStream
            move.w  d0,(a1)                ;Stick our data there
            bsr.w   Printf                 ;Print it (see routine below)
            ...

Pos.fmt     db      '%-d',0                ;Positive format string
Neg.fmt     db      '-%-d',0               ;Negative format string
```

Because the sign bit is the MSB in our data, separate
routines must be written to handle BYTES, WORDS, or
LONGS. This example above only works on WORDS.

Following is an entire library of 'C' type formatting
routines for assembly language programmers:

```
*****************************************************************
* NAME:     SPrintf()
* FUNCTION: Print formatted strings to a buffer.
* INPUTS:   A0 = Format string.
*           A1 = Ptr to arguments.
*           A2 = Ptr to buffer (preserved).
* RETURN:   None
* SCRATCH:  D0-D1/A0-A1
*****************************************************************

SPrintf     movem.l a2-a3/a6,-(sp)         ;Save stuff
            movea.l a2,a3                   ;Buffer
            lea     (PutChProc,pc),a2       ;Char fill procedure
            movea.l (AbsExecBase).w,a6
            jsr     (LVORawDoFmt,a6)         ;Do it!
            movem.l (sp)+,a2-a3/a6
            rts

PutChProc   move.b  d0,(a3)+                ;Char fill procedure
            rts

*****************************************************************
* NAME:     FPrintf()
* FUNCTION: Print formatted strings to a specified file.
* INPUTS:   A0 = Format string.
*           A1 = Ptr to arguments.
*           D0 = FileHandle.
* RETURN:   None
* SCRATCH:  D0-D1/A0-A1
*****************************************************************

FPrintf     movem.l d2-d4/a2-a3/a6,-(sp)   ;Save stuff
            move.l  d0,d4                   ;FileHandle
            lea     (PutChProc,pc),a2       ;Byte fill routine
            lea     (Buffer),a3             ;Ptr to your buffer
            movea.l (AbsExecBase).w,a6
            jsr     (_LVORawDoFmt,a6)       ;Do it!
            movea.l a3,a0
            bsr.b   WriteMsg
            movem.l (sp)+,d2-d4/a2-a3/a6
            rts

*****************************************************************
* NAME:     FPuts()
* FUNCTION: Writes a message to a specified file.
* INPUTS:   A0 = Ptr to message.
*           D0 = FileHandle.
* RETURN:   None
* SCRATCH:  D0-D1/A0-A1
*****************************************************************

FPuts       movem.l d2-d4/a6,-(sp)         ;Save stuff
            move.l  d0,d4                   ;FileHandle
            bsr.b   WriteMsg
            movem.l (sp)+,d2-d4/a6
            rts

*****************************************************************
* NAME:     Printf()
* FUNCTION: Print formatted strings to stdout.
* INPUTS:   A0 = Format string.
*           A1 = Ptr to arguments.
* RETURN:   None
* SCRATCH:  D0-D1/A0-A1
*****************************************************************
```

```
Printf      movem.l a2-a3/a6,-(sp)         ;Save stuff
            lea     (PutChProc,pc),a2       ;Byte fill routine
            lea     (Buffer),a3             ;Ptr to your buffer
            movea.l (AbsExecBase).w,a6
            jsr     (_LVORawDoFmt,a6)       ;Do it!
            movea.l a3,a0
            bsr.b   Puts
            movem.l (sp)+,a2-a3/a6
            rts

*****************************************************************
* NAME:     Puts()
* FUNCTION: Writes a message to stdout.
* INPUTS:   A0 = Ptr to message.
* RETURN:   None
* SCRATCH:  D0-D1/A0-A1
*****************************************************************

Puts        movem.l d2-d4/a6,-(sp)         ;Save stuff
            move.l  (_stdout),d4            ;stdout already open
            bsr.b   WriteMsg
            movem.l (sp)+,d2-d4/a6
            rts

*****************************************************************
* NAME:     WriteMsg()
* FUNCTION: Common subroutine used by FPrintf, FPuts, Printf, and Puts.
* INPUTS:   A0 = Ptr to message.
*           D4 = FileHandle.
* RETURN:   None
* SCRATCH:  D0-D1/A0-A1
*****************************************************************

WriteMsg    move.l  a0,d3                  ;How long is a piece of string?
1$          tst.b   (a0)+
            bne.s   1$
            exg     a0,d3
            sub.l   a0,d3
            subq.l  #1,d3                   ;Length
            move.l  a0,d2                   ;Buffer
            move.l  d4,d1                   ;FileHandle
            movea.l (_DosBase),a6
            jsr     (_LVOWrite,a6)
            rts
```

Note: All code used in this article is written in new
M68000 family syntax. The new syntax was developed
by Motorola specifically to support the addressing
capabilities of the new generation of CPUs.

☑

Are you a freelance Amiga Artist?

...Programmer?

...Musician?

...Project Manager?

...Toolsmith?

...Writer?

...Multimedia Consultant?

...Video Consultant?

...or any other Amiga specific freelance professional?

We are compiling a directory of international Amiga freelance talent! You get maximum exposure with your FREE listing! We want to hear from you!

In 200-250 words, describe your unique talents and services. You should also include a small list of recent projects.

Be sure to include your full name, company name (if applicable), full address, and phone/FAX numbers. Also, list any appropriate E-Mail addresses.

Send your information to:

PiM Publications, Inc.
Amiga Freelancers Directory
Attn: Listings
P.O. Box 869
Fall River, MA 02722-0869

or FAX 1-508-675-6002

For more information call 1-508-678-4200

WildStar

Discovering an AmigaDOS 2.0 Hidden Feature

by Bruno Costa

If you are a part-time MS-DOS or UNIX user, you have certainly typed a * instead of the AmigaDOS #? wildcard in your Amiga shell. Unless you were an ARP (AmigaDOS Resource Project) user you would have been prompted with something like "*.c: object not found". If you, like me, upgraded to Workbench 2.0, you were forced to drop ARP usage to benefit from the new and really improved operating system. This time there was no way out: you would really need to learn using the "#?" wildcard. And that is what you did — or at least what you tried to do. But (here is the but you were waiting for!) there is a solution, a solution already implemented by Commodore programmers that is hiding quietly, deep into AmigaDOS 2.0.

The solution to this wildcard compatibility problem was so simple, I was not surprised to see it working in less than five minutes (it was meant to be either a complete success or a complete failure). It involved simply the change of one single bit in the AmigaDOS root node—a global structure that holds some AmigaDOS system-wide variables. Studying the new include files provided for Workbench 2.0 programming, I noticed that the RootNode structure had, among others, a new field called rn_Flags which had only one bit with a defined meaning. This bit was defined by the symbolic name RNB_WILDSTAR—it surely meant either what I wanted it to be, or something really wild! I wrote a simple program to toggle this bit on, and after compiling, quickly typed a "dir *.c" to see that a new era in my Amiga Workbench 2.0 usage had begun.

Note that this feature was, to my knowledge, completely undocumented. It was not mentioned in my early version Workbench 2.0 "Using the System Software" manual or in any of the other manuals that came with my Amiga 3000. Even further, the bit definitions present in the include files didn't have any accompanying comments. Under such circumstances I don't have any explanations why this feature is not turned on by default (maybe a persisting bug ?), or why there isn't something like an environment variable that could be set to 1 or 0 to turn the feature on or off. I have been using this little command in my startup-sequence since the moment I discovered it worked. There is only one minor nuisance: when used alone in a pattern, such as in "copy * to ram:", the star will not mean what you would expect (copy all files in current directory to ram:), instead it will mean the current shell window (as it did when used under WB 1.3 or a default WB 2.0). Apart from that, I think this is a very welcome compatibility enhancement that works with all the AmigaDOS 2.0 commands that accept patterns.

```
/*
 * wildstar.c - toggles the star as a wildcard in AmigaDOS
 *
 * Bruno Costa - 29 Jan 91 - 29 Jan 91
 * (Compile with lc -L wildstar.c)
 */

#include <exec/types.h>
#include <clib/dos_protos.h>
#include <pragmas/dos_pragmas.h>

extern struct DosLibrary *DOSBase;

#define print(msg)   Write (Output(), msg, sizeof (msg))

void _main (void)
{
  struct RootNode *root = DOSBase->dl_Root;

  if (DOSBase->dl_lib.lib_Version < 36)
    print ("Sorry,  this program needs WB 2.0.\n");
  else
  {
    root->rn_Flags ^= RNF_WILDSTAR;

    if (root->rn_Flags & RNF_WILDSTAR)
      print ("The star is now a wildcard.\n");
    else
      print ("The star is now a normal character.\n");
  }

  _exit (0);
}
```

✓
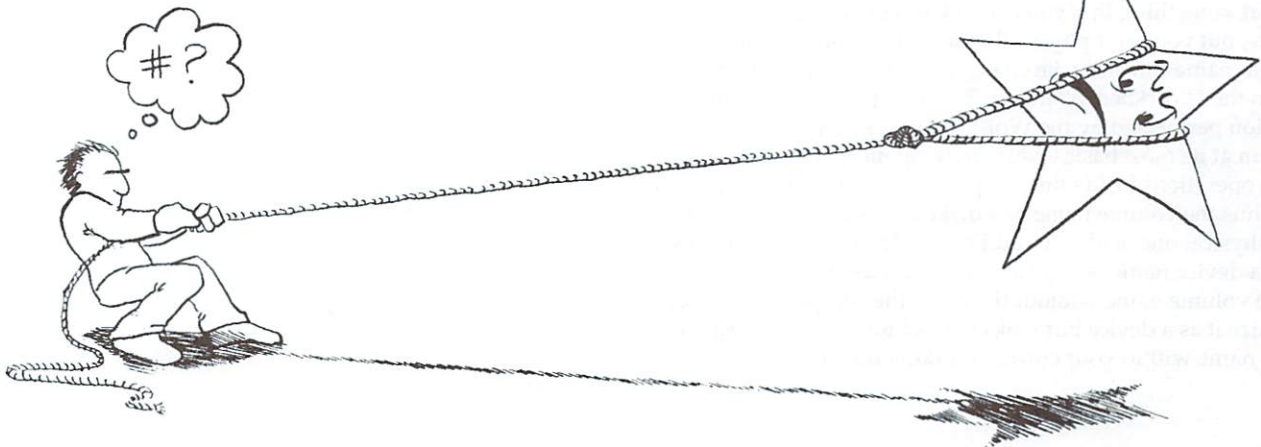
SYSTEM CONFIGURATION TIPS FOR SAS/C—ON DISK!

by Paul Castonguay

Last issue you saw three different installations of SAS/C for a two floppy Amiga, each one differing in how it used system resources to give you an environment in which you could write, compile, and execute programs written in C. Even a 512K system proved to be a very powerful C development workstation. In this document I will discuss those installations in detail.

BACKGROUND MATERIAL *(Skip if you like)*

DEVICE NAMES

A device name in AmigaDOS is any legal name whose last character is a colon. The name df1: is a device name and it represents a physical accessory connected to your computer, the external floppy drive. Thus df1: is an example of a PHYSICAL DEVICE NAME.

But the Amiga also has the ability to recognize other things through the use of device names. The easiest to understand is the volume name of a diskette. Suppose you have a diskette whose volume name is My_Work and it is mounted in df1:. To see what is on that diskette you could enter:

```
        Dir df1:
or
        Dir My_Work:
```

The second way refers to the diskette not by the physical device in which it is mounted, but by its volume name. That is an example of a LOGICAL DEVICE NAME.

Perhaps you are thinking that My_Work: should be called a physical device name because a floppy diskette is something physical, something that you can touch. Yes you can touch a diskette, but you can't physically touch its volume name. In fact, that name can easily be changed by using the "Rename" item on the "Workbench" menu. Renaming a diskette is an operation performed by the Workbench program, and a program at its most basic level is nothing more than a series of logical operations inside the computer's central processing unit. Thus the volume name of a diskette is a logical property, not a physical one, and since adding a colon on the end makes it also a device name, we get a "logical device name". If you use the volume name without the colon the computer does not recognize it as a device but looks instead for a file or directory of that name within your current working directory.

AmigaDOS stretches the concept of devices even further by allowing you to define a logical device name for any directory in your disk filing system. To do that you use the Assign command. Suppose in my previous example the diskette My_Work had two directories on it, C_Programs and BASIC_Programs. You could assign device names to each, like this:

```
        Assign My_C: My_Work:C_Programs
        Assign My_B: My_Work:BASIC_Programs
```

Now you can refer to these directories conveniently by their device names rather than their longer path names. The command:

```
        Dir My_C:
```

produces a report of the files stored in the directory My_Work:C_Programs. Again, the last character of a logical device name must always be a colon.

The Assign command uses the following format:

```
        Assign <device name> <directory name>
```

To see a report of the logical devices (as well as the physical ones) that are currently defined on your system enter the Assign command with no argument. Here is that report on my system:

```
Volumes:
My_Work [Mounted]
RAM DISK [Mounted]
RAD [Mounted]
Workbench [Mounted]

Directories:
My_C            My_Work:C_Programs
My_B            My_Work:BASIC_Programs
FONTS           Workbench:Fonts
S               RAM DISK:S
CLIPS           RAM DISK:clipboards
ENV             RAM DISK:env
```

```
T          RAM DISK:t
L          Volume: Workbench
C          Volume: Workbench
DEVS       Volume: Workbench
LIBS       Volume: Workbench
SYS        Volume: Workbench

Devices:
PIPE AUX SPEAK NEWCON DF1
PRT PAR SER RAW CON
RAM DF0 RAD
```

The report is divided into three sections: Volumes, Directories, and Devices. "Volumes" represent the names of the diskettes that are presently recognized by the system. They correspond to the disk icons appearing on the Workbench screen. "Directories" represent the logical device names that have been assigned to various directories in the disk filing system. You can see My_C: and My_B: at the top of the list. "Devices" represent physical devices attached to the system, printers, disk drives, and stuff like that.

TWO KINDS OF LOGICAL DEVICES

There are two different kinds of logical devices in the group called "Directories". The first is called "User Logical Device". My_C: and My_B: are examples of that. They were assigned by me, the user, for my own purposes. The other is called "System Logical Device". These are special reserved device names that have specific meaning to the computer. They are assigned automatically during boot-up. To be able to configure your Amiga yourself you must understand the purpose of at least four of these System Logical Devices.

C: DEVICE - COMMAND DIRECTORY

This is the System Logical Device in which all AmigaDOS commands are stored. On a generic Workbench diskette it is assigned to the Workbench:c directory. Whenever you enter an AmigaDOS command the system looks for it in the C: device, also called the command directory. If there is no program of the name you entered the computer responds with "Unknown Command".

S: DEVICE - SCRIPT DIRECTORY

This is the System Logical Device in which all script files are stored. Script files are text files containing AmigaDOS commands. The most important of these are the Startup-Sequence, StartupII, and Shell-Startup files that get executed automatically at boot-up time.

FONTS: DEVICE - FONTS DIRECTORY

This is the System Logical Device in which the Amiga's disk based fonts (fancy looking letters) are located. They have names like Diamond, Emerald, Sapphire, ... etc. They can be used to give your programs a more professional appearance. In this series of articles we will study how to access these fonts from our C language programs.

SYS: DEVICE - THE SYSTEM DIRECTORY

This System Logical Device serves the same purpose as the volume name on your boot-up disk. The computer needs to refer to the boot-up diskette often for its general operation, but that volume name can be changed by you. It therefore uses its own name, the logical device name SYS:, which is assigned automatically at boot-up time.

There are other System Logical Devices that are important for the computer's operation but I will not be doing anything with them in my discussion of system configuration. You can learn about them by reading any AmigaDOS reference book.

RE-ASSIGNING SYSTEM DEVICES

The most powerful feature of system devices is their ability to be re-assigned by the user. You can try this out yourself from a SHELL window. Enter the command:

```
Assign Fonts: RAM:
```

Then enter:

```
Assign
```

You now see that the Fonts: device points to the RAM disk. That means that software wanting to use the Amiga's disk based fonts will look for them at that location. But there is one problem, there are no fonts in the RAM disk. If you try now to run such software it won't be able to find them. You had better change it back.

```
Assign Fonts: SYS:Fonts
```

MAKING ROOM ON A WORKBENCH DISKETTE

I am often asked by beginners how to make extra room on a boot-up (Workbench) diskette. The usual reason for wanting to do that is to be able to run certain software without having to perform a multitude of disk swaps. In our case we want to use the SAS/C compiler and we do not want to have to flip diskettes every time we compile a program. At the same time we do not want to loose any important features that come on a standard Commodore Workbench diskette.

You should never just blindly delete things off your Workbench diskette. Often what you delete is important for the operation of your computer. Days or weeks later you may find that certain programs, or indeed certain features of certain programs, no longer work properly. By that time you will have forgotten what files you have deleted. What a mess!

SYSTEM RESOURCES ON TWO DISKETTES

A better approach is to distribute your Amiga's resources over two diskettes in such a way that the most frequently used ones are conveniently located on the boot-up one, while the rest are transferred to the other. You might call this second diskette "Workbench_2". But that's not all you should do. You should also make sure that the system knows where to find those transferred resources when it needs to use them.

→

For example, you might think that the fonts directory is not important enough to keep on your boot-up diskette. I agree. But if you blindly delete all the fonts they will no longer be available for software that needs to use them. Worse, you yourself will not be able to access them from within your C programs. A better approach would be to transfer your fonts to a second diskette and to inform the system where it can find them by assigning the FONTS: device to that location. By doing that you will both save room on your boot-up diskette, and still be able to use the Amiga's disk based fonts. When you run a program that needs to use them the system will ask you to:

```
"Insert Workbench_2" in any drive".
```

After your program reads whatever font it needs, you will be able to remove Workbench_2 and re-insert another diskette for the remainder of time that you use the program. Often the font will remain on the system after your program has terminated, allowing you to run the program again without having to re-insert the Workbench_2 diskette.

STEP BY STEP PROCEDURE
TO MOVE FONT DIRECTORY
To make sure you understand exactly what I am suggesting, let's work together and modify a standard (generic) work-bench diskette to use a font directory located on a separate diskette. To perform this experiment you will need one blank diskette and one unmodified copy of Commodore's Work-bench.

Step 1
Boot-up using a generic Commodore Workbench in df0: and a blank diskette in df1:

Step 2
Format (Initialize) the diskette in df1: calling it Workbench_2.

Step 3
Open a SHELL window and enter the following commands:

```
MakeDir df1:Fonts
Copy Fonts: to df1:Fonts all
Delete Fonts:#? all
```

Step 4
Enter the following command:

```
Ed S:Startup-Sequence
```

This opens the ED editor with the Startup-Sequence file.

Step 5
Using the [DOWN-ARROW] key move the cursor towards the end of the file and insert the following new line immediately before the LoadWB command:

```
Assign FONTS: Workbench_2:Fonts
```

Step 6
Press the [ESC] key followed by X and [RETURN]. This saves the file. Wait for all disk activity to stop.

Step 7
Re-boot and confirm that the system font device is now pointing to the Fonts directory on WorkBench_2 (use Assign command in the SHELL window). Also verify that your workbench diskette is now only 85% full (use the Info command in the SHELL window). Try using the fonts by executing the NotePad program and confirm that they are loaded from the Workbench_2 diskette.

Thus you have created room on your boot-up diskette without loosing the ability to use the Amiga's disk based fonts. When you boot up with this modified workbench the system will ask you to insert the Workbench_2 diskette in order that it can assign the FONTS: device to the correct directory. Insert it in df1: After boot up you can remove it. You will not need it until you run a program that uses the FONTS: device.

That completes the experiment.

There is an added advantage to transferring the disk based fonts to a second diskette. You now have enough room to combine with them the extra fonts that come on the "Extras 1.3" diskette, Courier, Helvetica, and Times. That will give you ten different fonts to use in your programs.

REMOVING COMMANDS FROM C: DEVICE
An often used trick to make room on a boot-up diskette is to remove AmigaDOS commands from the command directory. Again, if you are not careful you can get yourself into trouble, especially if you have little knowledge about what it is you are removing. Instead of just deleting commands you should transfer them to a second diskette and inform the system where it can find them in the event that they are needed, just like you did for the fonts. But in this case you will not be able to re-assign the C: device to Workbench_2. It must remain assigned to SYS:c where your most frequently used AmigaDOS commands are kept. Therefore, in order to inform the system of the location of transferred AmigaDOS commands, you will need to use a new concept.

SHELL ALIAS
Alias is a SHELL feature that allows you to redefine the names of commands and programs on your system. For example, suppose you are an experienced IBM-PC user and you want to use the word "Erase" in place of the AmigaDOS Delete command for removing files from your disks. You could

simply rename the Delete program in the command directory from Delete to Erase, but I don't suggest you do that. Someone else using your computer would not be able to figure it out. Besides, as you use your Amiga more and more, and your IBM less and less, you may find yourself wanting to use Delete instead of Erase. Another suggestion might be to make an extra copy of the Delete program in the command directory and call it Erase. That would work but it would also waste valuable disk space. What you really need here is a way to make both commands work on the same program and that is where the Alias feature comes in. It allows you to use a new name for an already existing command. Enter into a SHELL window:

```
Alias Erase Delete
```

The above Alias allows you to enter the command Erase and the computer will substitute in its place the command Delete. At the same time the command Delete still works normally. You can get a report of the alias definitions that are in effect on your system by entering the command Alias with no argument:

```
Alias
```

Unfortunately, the above Alias definition is effective only in your current SHELL window. If you open up another by double clicking on the SHELL icon you will find that the Erase command is unknown in that window. But there is a simple solution. You put your alias definitions in the file called "Shell-Startup". By doing that they will get established automatically at boot-up time and will be effective in every new SHELL window that you open.

ALIAS NAMES FOR COMMANDS
ON WORKBENCH_2 DISKETTE

It is possible to transfer certain commands from your command directory to the Workbench_2 diskette and to inform the system of their new location by giving them alias names. Using the same diskettes from our earlier font experiment, enter into the SHELL window:

```
MakeDir Workbench_2:c
Copy C:Ed to Workbench_2:c
Delete C:Ed
```

That copies the Ed editor to Workbench_2 and deletes it from the command directory. If you now try to use it by entering:

```
Ed test.txt
```

it will not work. But, if you enter the following command:
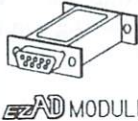
```
Alias Ed Workbench_2:c/Ed
```
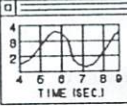
And try again:

```
Ed test.txt
```
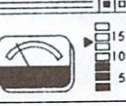
The system picks up the program from the Workbench_2 diskette.

Press [ESC], then Q, then [RETURN] to leave the editor. Note that if the Workbench_2 diskette is not physically mounted when you enter the above command the system will ask you to:

```
"Insert Workbench_2 in any drive"
```

So now you see that you can transfer commands from your command directory to another diskette and not loose the ability to use them. You do that by defining alias names that specify their entire path at the new location.

SMALL PROBLEMS WITH ALIAS

An alias works fine when you enter commands from a SHELL window, but not when you use the older CLI. You see, Alias is not an AmigaDOS command but a feature of the SHELL window, available only on version 1.3 of the Workbench. Thus to use this technique you will have to upgrade to AmigaDOS 1.3.

In addition alias names will not be recognized by the IconX program (running a script from an icon), although they will work fine if you execute that same script from a SHELL window. It turns out that this is a minor restriction because someone who runs scripts from icons usually has enough system memory to make those commands memory resident anyway, removing the problem. I will present the concept of making commands memory resident in a few minutes.

PRACTICE USING ALIAS NAMES

Since there is lots of room on the Workbench_2 diskette and since you will want to experiment a bit deleting different commands, why don't you copy your entire command directory to Workbench_2. Enter the following command:

```
Copy C: to Workbench_2:c
```

That way you won't have to bother making sure that you have previously copied commands to Workbench_2:c before deleting them from the command directory. They will all be there already.

There are two commands that you can safely delete from your command directory and transfer to Workbench_2, Ed and Edit. You can also transfer DiskChange and Lock if you do not have a 5 1/4 inch disk drive unit or a hard drive on your system. Simply add an alias definition in each case to the Shell-Startup file.

```
Alias Ed Workbench_2:c/Ed
Alias Edit Workbench_2:c/Edit
Alias DiskChange Workbench_2:c/DiskChange
Alias Lock Workbench_2:c/Lock
```

You can do the same with a number of other commands except that you may also have to modify the Startup-Sequence file as well (the file in the S: device that gets executed when you boot-up). Note that the Startup-Sequence file executes from a version 1.2 CLI command window, not a SHELL window, and as a result it does not recognize alias names. The solution is to modify the Startup-Sequence file to include the entire path name of any commands that have been transferred to the Workbench_2 diskette.

A good example of this is the SetPatch command. This command is used only once, during boot up. Thus there is little reason to leave it on your boot-up diskette. Delete it from your command directory and change the first line of your Startup-Sequence from:

```
SetPatch >NIL: ;patch system functions
```

to:

```
Workbench_2:c/SetPatch >NIL: ;patch system functions
```

For future reference you should also define an alias for it in the Shell-Startup file.

```
Alias SetPatch Workbench_2:c/SetPatch
```

Thus you have deleted a command from the command directory, transferred it to Workbench_2, and told the system where it can be located if needed in the future. Incidentally, if you have 1 Megabyte of chip ram on your system you should use the "r" argument in the above SetPatch command.

You can do the same thing with the AddBuffers command. Again you must change your Startup-Sequence from:

```
Addbuffers df0: 30
Addbuffers df1: 30
```

to

```
Workbench_2:c/Addbuffers df0: 30
Workbench_2:c/Addbuffers df1: 30
```

Then add an alias definition in the Shell-Startup file.

```
Alias Addbuffers Workbench_2:c/Addbuffers
```

Are you getting the idea?

ONE MORE WAY TO SAVE DISKSPACE FROM C: DEVICE

Yes, I still have another trick up my sleeve. However this one is only for owners of 1Meg or larger Amiga's because it uses up valuable system memory.

As you know AmigaDOS commands are stored in the C: device on the Workbench disk. Every time you use one the system must first load its corresponding program from the C: device on the floppy diskette in order that it can be executed. This makes the Amiga seem slow and sluggish compared to other computers. But that's not the way the Amiga was intended to be used. Instead, depending on how much memory you have available, you should transfer your most frequently used commands to system memory where they will execute instantly. Enter:

```
Resident C:Dir
```

From that point on whenever you enter the Dir command the system will use the version in system memory, not the one on the Workbench disk. In fact, the Workbench diskette no longer needs to be physically mounted in df0: for the Dir command to work.

To see a report of the commands currently resident in memory on your system enter the Resident command with no argument:

```
Resident
```

Here are the ones currently defined on my 2Meg system:

Name	UseCount
Assign	0
CD	0
Copy	0
Delete	0
Dir	0
Echo	0
Else	0

```
EndCLI      0
Endif       0
EndSkip     0
Execute     0
Failat      0
If          0
Info        0
List        0
Makedir     0
Mount       0
Resident    1
Run         1
Search      0
Skip        0
Sort        0
Type        0
Wait        0
```

As you can see, I use quite a few.

To automatically make AmigaDOS commands memory resident at boot-up time you must place instructions in the StartupII file, not the Startup-Sequence.

If you make a command memory resident it makes sense that you no longer need to keep a copy of it in your C: device. Remember however that if a command is deleted from C: the resident command in the StartupII file must refer to it by its complete path name, like this:

```
Resident Workbench_2:c/Dir
```

Thus the Dir command gets loaded into executable memory directly from the Workbench_2 diskette at boot-up time and no longer needs to be kept in the C: device.

In addition you must realize that during boot-up the Startup-Sequence file knows nothing about these resident commands. This is a similar problem to the one described above where the Startup-Sequence did not recognize alias names. The solution is the same, modify the Startup-Sequence file to include the complete path name of any command that it uses from the Workbench_2 disk.

A good example is the Echo command. Suppose you delete it from your command directory. The resident command in the StartupII file must specify:

```
Resident Workbench_2:c/Echo
```

and the Startup-Sequence file which uses the Echo command must use its full path name:

```
Workbench_2:c/Echo "Amiga Workbench Disk. Release 1.3.2 version 34.28"
```

Summary

I have presented three ways of making extra room on your boot-up diskette without losing any of the features normally available on a Commodore Workbench diskette:

1. Transfer a system logical device to a second diskette and re-assign that device to let the system know its new location.

2. Transfer AmigaDOS commands to a second diskette and use alias names to let the system know their new location.

3. Transfer AmigaDOS commands to a second diskette and make them memory resident by loading them directly from their new location.

More information on the installation of SAS/C is contained on the enclosed disk.

✓

Hash for the Masses

An Introduction to Hash Tables

by Peter Dill

Nicholas Wirth, creator of the computer languages Pascal and Modula-2, titled one his books on programming *Data Structures + Algorithms = Programs*. Surely this title is one of tantalizing simplicity for someone trying to decompose a programming problem into these manageable units. There is a definite interplay between algorithms and data structures—some algorithms place heavy emphasis on rapid data insertions and queries, while for others it is more important to quickly find data items that are related lexigraphically. With enough work by a programmer, any data structure will allow access to its data in whatever manner is desired; however every structure has certain patterns of access in which it excels. The Damocles' Sword that hangs over all embryonic programs is the fear that the need to manipulate data in non-optimal ways will crop up and results will be ugly and worse, slow.

Choosing a data structure which performs poorly for the demands that are made upon it is to risk a program that runs too slowly to be useful. To avoid this result, one must either modify the algorithm or change the data structure. In most cases computer programming involves negotiating the incompatibilities between the most obvious algorithm and the most natural data structure. With this in mind, it is important to have a range of data storage methods available and to know their strengths and weaknesses.

A hash table is a storage scheme which excels in quick deletions, insertions, and queries, and is a good match for programs that make heavy use of these functions. This article gives an overview of the concepts behind hashing, examines the situations were it performs well and those where it does not, and gives code for a standard library which will allow any user program to incorporate this data structure with just a few function calls.

An Easy Example

The choice of a data structure for a program is influenced primarilyby the nature of the data being stored and the way it will be accessed. Consider the following venerable computer science pedagogical problem:

Write a program letters.c which determines the number of occurrences of each alphabetical letter in the input. Output should be a letter paired with the frequency of its presence in the input data.

Of course one way to solve the problem is to have a lot of statements like *if (input_ch == 'A') Num_A++;*. Intuitively, there must be variables which will keep track of the number of each letter observed so far, but to have 26 variables and 26 *if* statements makes for an unnecessarily lengthy and cluttered program.

One obvious optimization tactic is to use an array with 26 elements to hold the information about each letter—perhaps something like:

```
struct letter_info

char letter;
short occurrences;
;
struct letter_info table[26];
```

But which letter should go in *table[0]*? Either we could put each letter in the order it first occurs in the input, or we could assign the letters to the array indexes using the ordinary ordering: 'A' to 0, 'B' to 1 and so on. If we opt for the first method, then to update the information for a particular letter we would start at *table[0]* and loop through all the elements until we got to one that is storing the letter we are interested in. This method is similar to the method with many *if* statements in that we rummage around to find the storage spot we need. The advantage is that now we need only one array and a short loop to accomplish the same thing.

The idea of using the normal alphabetical ordering to position the letters in the array is more interesting still. It almost seems redundant to store *table[0].letter = 'A'* because we know that the first letter will always be the first item. Even worse is the idea of looping through the array to find the information for 'D' when we know that it will be in the fourth slot. What we need is a way to turn this association into something that the compiler can understand.

To do this, we use the fact that letters are given a numeric ordering in C by their ASCII codes. These codes aren't going to be in the same range as the array indexes, but by converting them, we can move freely between the idea of an alphabetical character and its position in the data structure. Figure 1 shows an implementation of letters.c that uses this

idea to reduce the whole program to a few lines. Notice that the program goes right to the array element it wants without searching to find it.

Variation on an Easy Example

A slight variation on *letter.c* just considered is the following problem:

> Write a program *words.c* which keeps track of how many times individual words in the input file occur. The program should also keep a list of the pages in the file on which a word appears.

The algorithm for this program is as simple as the one for *letters*. Take the current word in the input file to see if it is already being stored. If it isn't, then tell the data structure to add it with *occurrences = 1*. If it is already present, then just add one to the frequency count. The current page number is determined by the current line of the input file.

For each word we will have to have a structure that keeps track of a number of things. We need a string to store the word itself. This field is known as the key field and is what the data structure will be organized by. The data structure will be geared to performing actions using the key field—find a record with a given key; insert a record with this key; delete a record with this key. Notice that we will never be doing something like looking for all the words that appear on page 48. The fact that we will be accessing data by only one key field will be important in choosing a data structure.

Also we will need an integer to represent the number of times the word has occurred, and a list of integers that represents the pages that contain occurrences of the word. Some words might be in the file only once and so will have only one page in their list. Common words, however, probably appear on every page and consequently would have a long list. Because the number of pages will vary so widely, it makes sense to store the pages in a linked list so as to use only as much space as necessary.

Likewise, because the number of items to be stored can't be determined until the program is actually run, it is necessary for the program's main data structure to be dynamic as well. A dynamic structure, such as a linked list, has its maximum number of elements limited only by the amount of memory available and can add new items while the program is executing. This contrasts with static structures such as arrays where the maximum storage space is fixed when the program is compiled. Notice that in *letters.c* a fixed array size wasn't an obstacle because we had prior information about how many items would have to be stored.

In most cases it is not possible to find items in a dynamic data structure by selection so a traversal system must be used. Selection is a method of data access where we compute the subscript of the array element we are interested in and hence are able to go directly to it. By contrast, in traversal the structure elements are considered one at a time until the appropriate item is found. In the final version of *letters.c*, we were able to use selection to locate array elements. In order to determine if a given word is in the structure in a linked list that is ordered alphabetically, we follow a traversal procedure that searches from the beginning of the list until it finds the key (a hit) or a word that is lexigraphically greater, in which case it is known that the key word isn't present and the search can be aborted (a miss).

Considering that almost all of the execution time of *words.c* will be the result of the time taken by operations on the program's main data structure, it is important to examine how a linked list will perform for an average query operation.

For a given key word, it is possible that we might find it at the beginning of the list, we might have to search all the way to the end, or we might be able to stop the search somewhere part way into the list. Over a large number of words, it can be expected, that we will be able to discontinue the query after checking half the list. So if we call the time taken to check a single record c_1, and let s be the length of the list (one record for every different word in the input file), then time taken by *words* to process a file with n many strings will be roughly $n \times (c_1 \times s/2)$. Of course, as the program loops through the input, it will be expanding the list size s so this is only an approximation; nevertheless, we can make some useful generalizations. If *words* were run on an on-line book that had a half-million words, 10% of which form the vocabulary for the book, then the program would perform something on the order of 12.5 billion key checks; even assuming that c_1 is small this could take a long time to execute. Since we can't change n, obviously, to reduce the running time we need to reduce the time taken to query the data structure to as small a number as possible.

Query time, then, is the principle concern in choosing a data structure to implement the algorithm for this program. In *letters.c* a method of interpreting the index of the array was used which allowed the data to be accessed with no searching at all. The length of time taken to access the appropriate record was basically the length necessary to convert the key into an array index; this characteristic is known as "constant time" as the queries always take some constant length of time regardless of how much data is being stored. A constant time query data structure is important because no penalty is paid as the data size increases—in contrast to a linked list implementation of *words* discussed above, a hypothetical constant time version would need only half-a-million key comparisons. So if c_2 is the time taken by a single constant time query, and if $c_1 \approx c_2$, then this new program would be approximately 25,000 times faster.

Apparently, a linked list would not be a good choice for the main data structure for *words.c*. Unfortunately, it is not obvious how to turn an array into a constant time query data structure which will fit this program. The first problem is how to turn a key word into an array index. The instance where the key is just a single letter is almost trivial; when the key is a string of characters things become more challenging.

Secondly, assuming that we take the first 12 characters of the key to be significant, there are over $\$52^{12}$ possible keywords. Notice in *letters.c* there was a one-to-one correspondence between the number of permutations of the key and the size of the array. Now, however, it would clearly be absurd to

```
/* letters.c */
#include <stdio.h>
#include <ctype.h>
  char table[26];
main()
  {
    int i, ch;
    while ((ch = getchar()) != EOF)
      if (isalpha(ch))
        table[toupper(ch)-'A'] += 1;
    for (i=0; i<26; i++)
      printf("%c: %hd\n", ('A' + 1), table[i]);
    exit(0);
  }
```

Hashing Functions

The term hash function is used to refer to both the function that converts a key into an integer and the function that converts that a key into a integer in the range of the array index. For the sake of clarity I will refer to the first function as $g()$ and the second as $h()$. Obviously $h(k) = g(k)$ % *size* where *size* is the number of indices for the array and % has its usual meaning as the modulus division operator in **C**. Of course the plan is to store the record with key k at *table[h(k)]*.

A hash function should satisfy the following criteria:

• Should spread values over the full range.

• Should be possible for short string to hash to large values.

• Should be relatively quick to compute.

• Should the preserve distinct nature of permutations alphabetical keys—"file" should hash to a different value than "life".

• Shouldn't have a range with a disproportion of multiples of any number.

• Shouldn't give alphabetically similar keys similar hash values.itemize

Using the fact that characters are given a numeric value by their ASCII codes, we can essentially deal with a key string as a list of up to 12 numbers. The challenge is to turn this group of numbers into a single number in a way that meets the goals listed above. Notice that multiplying them together isn't a good idea as anagrams will get the same value, and it will not be possible to hash to a prime. The following is a modification of the method presented in Sincovec and Wiener's Data Structures Using Modula-2:

allocate 52^{12} array elements, especially considering that our knowledge of the input data tells us that we will most likely need to store a few ten-thousand records.

There is a theorem of combinatorics, known as the Pigeonhole Principle, which states that in trying to put $n + 1$ object into n pigeonholes that some pigeonhole is going to get more than one object. Knowing that in dealing with a theoretically huge data set of English words and the much smaller array size that can physically be allocated for data storage, it will be necessary to assign some of the possible key permutations to a single array index; the second challenge is what to do when two such words actually occur in the input file.

When we have methods for converting a string into a valid array index and for handling situations where more than one word key is assigned the same index, the resulting array is known as a hash table. Using this modified array, we will be able to insert, delete, and query items in constant time. It will turn out that some hash tables are also dynamic and hence a prefect fit as data structures.

Figure Two
Breaking "throughout" into clusters

T	h	r	o	Cluster 1
u	g	h	o	Cluster 2
		u	t	Cluster 3

Treating each letter as a 7-bit integer, break the key word up from the left into three clusters of four letters each (see figure 2).

Shift all the bits in Cluster 2 to the left, one space.

Shift all the bits in Cluster 3 to the left, two spaces.

XOR clusters 1, 2 and 3 together. The result is $g(k)$.

ASCII is a seven-bit code which allows us to treat the letters as seven-bit integers without losing any information. Notice that the clusters are all less that 32 bits long and hence they fit into one

68000 series register. Because all the manipulations take place in CPU registers, the hash value can be computed quickly. Using this hashing function, I have found that in practice, regardless of the number of items stored, the average length of a non-empty list is about 1.5 for a table with 1-to-1 ratio between its size and the number of elements in it. This constant time performance insures that queries will be quick.

Collision Resolution

To insert a record into the hash table, we determine the hash value of the record's key. If this array element already contains a record, the situation is known as a collision and the method of determining where to store the new record is known as the "collision resolution policy." Essentially there are two fundamental methods of proceeding—open addressing where the record is placed at some other index in the array, and separate chaining, the most generally useful scheme, where additional records are placed outside of the array.

In separate chaining each array element represents a linked list of records which have the same hash values. Note, however, that these records don't have the same key. In performing a query on a given key, first the hash value is computed and then the linked list at that table index is searched in linear order until either a record with the desired key field is found or the end of the list is reached. If the end of the list is reached without finding a record with the sought key field, then it is known that record cannot be stored anywhere in the table.

The dispersion of keys created by $h(k)$ is critical to the performance of the hash data structure. The absolute worst case is if every key hashes to the same value—the access time will be similar to that of a linked list. Conversely, in the best case there is only one record stored at each index and so finding an element in the table will only take as long as it takes to compute $h(k)$. If the hashing function gives an even distribution, and given a table of size *size* containing *s* records, then we would expect each table slot to have *size/s* elements. So, given a judicious choice of the array size, access time will depend entirely on $h(k)$, and consequently hashing will give the constant time queries that we desire.

In an array, the entire data structure is stored in contiguous memory, which in effect means that the absolute memory location of any element can easily be determined by knowing the initial array location and the size of the objects stored (remember all elements are allocated the same amount of storage space). The memory location of element *i* is simply *&table*[0] + *i* X *sizeof(table*[0]). The downside of this easy access is that the maximum size of the array must be specified at the time the program is compiled. As was seen above, while a hash table is never really full, it is advantageous to have the size of the table in a one-to-one ratio with the number of elements being stored. To help accomplish this, *hashlib* dynamically allocates the hash table using the *calloc()* function rather than statically allocating it at compile time. And while it is still the case that the size of the table must be chosen before any items are stored in it, in practice it is often possible to determine during execution (perhaps through the use of a command line switch) the maximum number of elements to be stored.

Obviously, the time taken by a query will depend upon the length of the linked list that is followed after computing $h(k)$ and hence the hash table size will have an effect on performance. However, the numeric nature of the size will also have an effort on performance. Suppose that a certain table has an even number of entries, and suppose that $g(k)$ produces all even values either because of a defect in the function or because the input data just happens coincidentally to be skewed with words that hash to even numbers. Then the values of $h(k)$ will consequently be even and all the odd table indexes will never be used so the average length of a linked list will be doubled and therefore access time will be increased. In general if n is the size of a hash table which has prime factors $a_1,...,a_k$ and if for some key k, $g(k)$ has $a_i,...,a_j$ among its prime factors, then $h(k)$ will be a multiple of a_i X ... X a_j. So if there is a numeric artifact in the range of $g(k)$, then $h(k)$ will not distribute all indexes equally and there will be a corresponding denigration in access time. In practice this means that prime tables sizes that are optimal.

Standard Libraries

Unfortunately the complexity of a program does not increase linearly in relation to the code size—rather it is the case that twice the code is likely to be an order of magnitude harder to understand. To combat this, a programmer must be rigorous in planning a project and in using proven methods to help control program complexity. The use of high-level languages, modular code, careful documentation, and standard libraries and interfaces can be of great help in keeping complicated code understandable.

Standard libraries offer a number of benefits to the programmer. They aid debugging by removing large sections of presumably proven code from cluttering the new, task-specific code. They also help by substituting a one-line, hopeful self-descriptive function call for an inline alternative; consequently, user functions are succinct and their purpose can be grasped mores easily. Furthermore, a standard library encourages the use of more sophisticated and efficient data structures by making their ease of use comparable with that of structures with built-in language support like arrays. This eliminates "quick and dirty" attempts to try out an algorithm which latter requires a rewrite for a different data storage method to make them practical.

The hashlib Library

Hashlib.c is an implementation of a hash table and the access functions associated with it that I have described in this article. It is an abstract version of a hash table in the since that it doesn't need to know about the kind of information the user is storing. So whether the routines are used with a program that stores payroll information or one that manipulates publishing data, the code in *hashlib.c* doesn't have to be altered. To accomplish this the library uses two header files and requires the user to provide three simple functions.

The structure that the user program wants to organize in a hash table may be simple and consist of only a key field, or it might be very complex. However, for the purposes of

➤

hashlib.c, there are only a few things about the structure that are important. First there must be a way to refer to the structure. The library expects the structure to be called by the typedef *element*. This definition is then put in a header file *userdefs.h* and is read at compile time.

The library has no idea which field in an *element* the user has picked to be the key field and this is desirable because for some applications it might be *employee_name* and for others, *BookTitle*—to hard code a name would be too restrictive. To allow *hashlib.c* to access this field without having to know its name, the user writes a short function *retkey()* which takes as input a pointer to an element and simply returns the value of whatever field the user has decided to organize the table by. Similarly, copying the contents of one *element* to another is done by another user-supplied function, *copyelement()*. Both of these functions can be placed in any source file as long as they are linked with the library code.

Sometimes it is useful to have dynamic data structure, prehaps a linked list, perhaps a gadget, as part of an *element*. Of course, this structure must return its memory when the *element* is deleted. This is done by the function *deleteelement()* which the user writes and places in one of the source files. If an *element* contains no dynamic memory, then this function can simply be a stub.

It has been my experience that it is extremely difficult to remember the correct interface required for a standard library function call (for instance, what is the order of arguments for *qsort()*), so I have tried to make the call format for the functions as simple as possible. To this end, I have eliminated the need for the user to keep track of a variable for the hash table by making it a global variable. Consequently, most of the calls involve only one variable and hence the danger of confusing call syntax is vastly reduced. On the downside, this prevents the use of more than one hash table in a program but this shouldn't present a difficulty for most applications.

The following are the call syntax is for the hash table functions:

short makehash(long)
The input value is the size to make the newly created hash table. The return value is FALSE if memory isn't available.

element* inserthash(element*)
The input value is a pointer to a *element* to copy into the table. If the new item is successfully inserted, a pointer to it is returned. In the event of an insertion failure, NULL is returned.

element* findhash(char[])
The input value is a the key string to check the hash table for. If a record with a matching key field is present, a pointer to it is returned. A NULL pointer signifies that the item is not in the table.

short deletehash(char[])
The input value is the key string for a record to delete from the table. If the record isn't present, then FALSE is returned. It calls *deleteelement(*element)*, which is defined in the user's program, to delete the dynamic memory, if any, that is associated with an element.

element* dumphash(char[])
This function is used to get sequential access to all the records in the table. The return value is a pointer to the record stored after the one with the key field that is passed to the function. An input value of NULL causes *dumphash()* to return a pointer to the first record and is used to start looping through the data. A return value of NULL means that all values in the table have been returned.

void freehashtable()
Because the memory for the hash table is dynamically allocated, it is necessary to use *free()* to return it to the system when it is no longer needed. This function frees all the linked lists that are created when new items are inserted.

To summarize the following need to be done by the user when accessing the library:

1. Create the file "userdefs.h" contains the definition typedef element which is user data structure.

2. Write a function *copyelement()* which copies arg2 to arg1, where the arguments are of type **element* and place it in a source file.

3. Write a function retkey() which returns the key field associated with the argument **element* and place it in a source file.

4. Write a function *deleteelement()* which uses *free()* to deallocate any dynamic memory associated with an *element* and place it in a source file. Input is a pointer to an element.

5. Include " hashdefs.h" and "userdefs.h" in all code files that access the hash library.

6. In the main program call *makehash()* with an argument specifying how many records are expected. Before the program exits call *freehashtable()*.

7. Compile *hashlib.c* and the user program with long integers.

8. Link the object files together.

You might notice when examining *hashlib.c* that the function to compute the hash value of a key is split up into two macros. The use of a macro avoids the overhead of a function call and consequently runs about 60% faster. However, some compilers have a limit of 256 characters in macros, so I have split *computehash()* up into two #defines to provide compatibility with various compilers.

Draw Backs of Hashing

The value of a data structure depends on how well it fulfills the demands made on it in a program. As we have seen a hash table is a method of data storage that has constant time queries, insertions and deletions. There are some actions which hash tables don't support well though. Because hashing does not preserve the lexigraphical ordering of the input keys, it is not possible to do wildcard searches and the useful relations that exist between contiguous elements in trees aren't there in the table. Hashing is not used to sort—its value is in programs with algorithms that make heavy use of queries on large data sets.

There is also memory overhead associated with using a hash table. For every slot in the table there is a four-byte pointer to the next value. And for additional items in the list there are more pointers. The memory efficiency of the data structure depends on a good hashing function to disperse the keys and keep the collision lists short.

Twists on Hashing

There are collision resolution schemes quite different from the one described here where only slots in the original array are used to store elements. In some methods the new item is placed in the next available spot and there are others in which a second hashing function is used to pick another space. Using this kind collision scheme complicates finding and deleting items, but it can be useful in situations where it is desirable to avoid allocating memory dynamically.

Another interesting use of hashing is as a space-saving compression scheme. The main idea here isn't to store items in the array by a numeric index, but rather to use the hash value as a shorthand for the actual key. To do this, we compute $g(k)$, for a key k and then store it instead of the actual key. Because $g(k)$ is a long integer which is only four bytes and k is a 12-byte string, there is an eight-byte savings for every key. Of course now it is impossible tell which key we are really storing because we only have the hash value, but for some applications this is acceptable.

Algorithms and Data Structures

Almost any data structure can be used with a given algorithm. Arrays, dynamic linked lists, AVL trees, heaps and hash tables all essentially do the same thing: store data and give it back again. However just like thumbtacks and staples they have certain jobs they do best. Picking the wrong tool, like a thumbtack to hold together a report or a hash table to sort a list of records, can have awkward if not painful results. There are many situations where a hash table will not be flexible enough to be of use—but in those instances where it is appropriate, it is usually the best tool for the job.

words.c

```c
/* words.c
    A program using hashlib.c to keep a count of the number of occurrences
    of words in a text file and the pages on which they occure.
*/

#include <stdio.h>
#include <ctype.h>
#include "userdefs.h"
#include "hashdefs.h"
#define FALSE 0
#define TRUE  1

#define PAGELENGTH 80
#define STRLENGTH 19

extern char* malloc();

char* retkey(data)
  element   *data;
  {
    if (data == NULL)
        return NULL;
    else
        return data->word;
  }

  /* note that there is no need to copy page lists */
void copyelement(one, two)
  element   *one, *two;
  {
    (void) strcpy(one->word, two->word);
    one->occurrences = two->occurrences;
    one->start_page_list = NULL;
    one->end_page_list = NULL;
  }

  /* Used to free the linked list of page numbers that is part of an
    element. */
void deleteelement(data)
  element   *data;
  {
    struct plist   *cur, *next;

    if ((data == NULL) || ( (cur = data->start_page_list) == NULL))
        return;

    for (next = cur->next ; next != NULL; cur = next, next = next->next)
        free ((char *) cur);
    free ((char *) cur);
  }
```

```
short copyword(pos, one, two)                              main(ac, av)
  char  one[], two[];                                        int   ac;
  short *pos;                                                char  *av[];
  {                                                          {
    short i;                                                   FILE   *text_file;
                                                               char   buffer[250], word[250];
    if (two[*pos] == '\0')                                     short  line, page, pos;
        return (FALSE);                                        element *info, entry;
    while (!isalpha(two[*pos]))                                long   length;
        if (two[++(*pos)] == '\0')
            return (FALSE);
    for (i = 0; two[*pos] != '\0' && isalpha(two[*pos]) && i < STRLENGTH;
        (*pos)++, i++)
        one[i] = tolower(two[*pos]);                          if (ac != 3)
    one[i] = '\0';                                            {
    return (TRUE);                                               fprintf(stderr, "USEAGE: %s <text file> <length>\n", av[0]);
  }                                                              exit (5);
                                                             }

/* Adds a new page to the page list for the word stored in info. */
void add_occurance(info, page)                               if ((text_file = fopen(av[1], "r")) == NULL)
  element *info;                                              {
  short    page;                                                 fprintf(stderr, "Could not open file %s.\n", av[1]);
  {                                                              exit(5);
    struct plist   *temp;                                    }
                                                             sscanf(av[2], "%ld", &length);
    if ((info->end_page_list != NULL) && (info->end_page_list->page == page))
        return;                                              if ( makehash((long) length) == FALSE)
                                                             {
    if ( (temp = (struct plist*) malloc (sizeof(struct plist))) == NULL)   fprintf(stderr, "Could not make hash table of size %ld.\n", length);
      {                                                          exit (10);
        fprintf(stderr, "Out of memory!\n");                 }
        exit(10);
      }                                                    /* Loop through all the input updating data information as
    temp->next = NULL;                                          necessary.*/
    temp->page = page;                                     for(page=1, line=1; fgets(buffer, 250, text_file); line++)
                                                             {
    if (info->start_page_list == NULL)                          if (line == PAGELENGTH)
        info->start_page_list = temp;                           {
    else                                                            line = 1;
        info->end_page_list->next = temp;                           page++;
    info->end_page_list = temp;                                 }
  }                                                            for (pos = 0; copyword(&pos, word, buffer) ; )
                                                               {
void printpages(word)                                              if ( (info = findhash(word)) == FALSE)
  element *word;                                                   {
  {                                                                   entry.occurrences = 0;
    struct plist   *cur;                                               (void) strcpy(entry.word, word);
                                                                       info = inserthash(&entry);
    for (cur = word->start_page_list; cur != NULL; cur = cur->next)    }
        printf(" %hd ", cur->page);                                (info->occurrences) += 1;
    printf("\n");                                                 add_occurance(info, page);
  }                                                              }
                                                             }

                                                           /* Display all the accumulated data. Notice the initial and termination
                                                               conditions used with dumphash(). */
                                                           for (info = dumphash(NULL); info != NULL; info = dumphash(info->word) )
```

```c
      {
        printf("%s (%hd) ", info->word, info->occurrences);
        printpages(info);
      }

    fclose(text_file);
    freehashtable();
  }
```

HashLib.c

```c
#include <stdio.h>
#include "userdefs.h"
#include "hashdefs.h"
#define FALSE 0
#define TRUE  1

/*  HASHLIB.C
    Peter Dill

    HASHLIB.C is a collection of functions to define, create, and manipulate
a generic hashtable. A hash table is an array whose indexes are related
to the data stored there. This allows an element to be found in the time
it takes to compute the index (called hashing) regardless of the number
of items stored. This is contrast to a binary tree where the time to find
an item can be log(base 2)n or a linked list, n.
    Different keys can generate the same hashindex, so when a collision
occures a linked list is made. So if the ratio of hashtable slots to
items stored rises, preformance denegrates toward a linked list case.
    The memory for the hashtable here is calloc'ed so its size is determined
by calling program.
    The hashtable is generic in the sence that the type of items it stores
is determined by the definition of 'element' that the user provides in
"userdefs.h". The hashtable contains pointers to a structure called
'packet' which contains an 'element' and a pointer to another 'packet'.
This allows for a linked list. The definition of 'packet' is in
"hashdefs.h".

For a program to interface with the functions four things are needed:

1. external typedef needed for 'element' which is the type for the user
   data. Placed in "userdefs.h"
2. retkey() is an externally defined function which returns the key field
   of an instance of '*element'.
3. copyelement() copies arg2 to arg1 where the args are of type element.
4. deleteelement() which takes a pointer to an element and deletes the
   dynamically allocated memory, if any, associated with it.
These functions are placed in a file with the rest of the user code.

Remember to compile and link with long integers.
*/

extern char* malloc();
extern char* calloc();
```

```c
long TableSize;
struct packet **ht;


#define computehash1(k,h)\
for (;(k[ii] != '\0') && (ii < 4); ii++)\
  {\
    c1 <<= 7;\
    c1 |= k[ii];\
  }\
for (;(k[ii] != '\0') && (ii < 8); ii++)\
  {\
    c2 <<= 7;\
    c2 |= k[ii];\
  }


#define computehash2(k,h)\
for (;(k[ii] != '\0') && (ii < 12); ii++)\
  {\
    c3 <<= 7;\
    c3 |= k[ii];\
  }\
c3 <<= 2;\
c2 <<= 1;\
h = (c1 ^ c2 ^ c3) % TableSize;


void deletechain(lst)
  struct packet *lst;
  {
    struct packet *cur, *next;

    if ( (cur = lst) == NULL)
        return;
    for (next = cur->next; next != NULL; cur = next, next = next->next)
      {
        deleteelement(&(cur->data));
        free (cur);
      }
    deleteelement(&(cur->data));
    free (cur);
  }


struct packet* makepacket(dt)
  element *dt;
  {
    struct packet *ptr;

    ptr = (struct packet*) malloc ((long) sizeof (struct packet));
    if (ptr == NULL)
      {
        puts("\n could not allocate memory");
        return(FALSE);
      }
    ptr->next = NULL;
    copyelement(&(ptr->data), dt);
    return(ptr);
  }
```

```c
/* input is the size for the new hash table. The return value is true if
   the table was sucessfully made. It is desireable to have the size
   equal to the maximum number of elements to be stored in the array.
*/
short makehash(size)
  long size;
  {
    long   i;

    if ( ((size >>1) <<1) == size)
        size += 1;      /* change to an odd */
    TableSize = size;   /* global, user wont have to keep passing the value*/
    ht = (struct packet**) calloc(size, sizeof(struct packet*) );
    if (ht == NULL)
      {
        fprintf(stderr,"\n Could not allocate memory for a  %d table",
            TableSize);
        return(FALSE);
      }
    if (ht[3] != NULL)
        for (i=0; i < TableSize; i++)
            ht[i] = 0;  /* make sure the pointers are zeroed just in case */
    return (TRUE);
  }


/* Input is a pointer to an element. If record with a same key as that of
   the input element is already in the table the item will not be added.
   Otherwise a new record is created with the same field as those of the
   input and is added to the table at the end of a list. A pointer to this
   new record is returned. NULL indicates a failure.
*/
element* inserthash(dt)
  element *dt;
  {
    register unsigned long  c1 = 0, c2 = 0, c3 = 0;
    register short ii=0;
    char   *key;
    unsigned long   hashindex;
    struct packet *ptr, *tmp;

    key=retkey(dt);
    computehash1(key,hashindex);
    computehash2(key,hashindex);
    for (tmp = ht[hashindex]; (tmp != NULL); tmp = tmp->next)
        if (strcmp(key, retkey(&(tmp->data))) == 0)
            return (NULL);          /* duplicate element */
    ptr = makepacket(dt);
    if (ptr == NULL)
        return (NULL);
    ptr->next = ht[hashindex];
    ht[hashindex] = ptr;
    return &(ptr->data);
  }
```

```c
/* Deletes the record with the key given by the input parameter. If
   no such record exists then FALSE is returned.
*/
short deletehash(key)
  char  key[];
  {
    register unsigned long  c1 = 0, c2 = 0, c3 = 0;
    register short ii=0;
    unsigned long     hashindex;
    struct packet **ptr, **parent;

    computehash1(key,hashindex);
    computehash2(key,hashindex);
    if (ht[hashindex] == NULL)
        return(FALSE);
    parent = ptr = &(ht[hashindex]);
    while ( (*ptr != NULL) && (strcmp(key, retkey(&((*ptr)->data))) !=0) )
      {
        parent = ptr;
        ptr = &((*ptr)->next);
      }
    if (*ptr == NULL)
        return(FALSE);
    if (parent == ptr)                  /* item sought is the at the head */
      {
        ht[hashindex] = ht[hashindex]->next;
        free(*ptr);
      }
    else
      {
        (*parent)->next = (*ptr)->next;
        free(*ptr);
        *ptr = NULL;
      }
    return(TRUE);
  }


/* Returns a pointer to the element that has the key given in the input
   parameter. If no such item is in the table, NULL is returned.
*/
element* findhash(key)
  char  key[];
  {
    register unsigned long  c1 = 0, c2 = 0, c3 = 0;
    register short ii=0;
    unsigned long     hashindex;
    struct packet *ptr;

    computehash1(key,hashindex);
    computehash2(key,hashindex);
    if (ht[hashindex] == NULL)
        return(NULL);
    ptr = ht[hashindex];
```

```c
    while ( (ptr != NULL) && (strcmp(key, retkey(&(ptr->data))) != 0) )
      {
        ptr = ptr->next;
      }
    if (ptr == NULL)
        return(FALSE);
    return(&(ptr->data));
}


/* Used to get access to every item in the table sequentially. Returns
   a pointer to the element which follows in the table the record with
   the key given by the input parameter. To start pass the value NULL.
   A return value of NULL from the function indicates that all the input
   parameter was the key for the last item in the table or it wasn't
   present.
*/
element *dumphash(last)
  char  last[];
{
    register unsigned long  c1 = 0, c2 = 0, c3 = 0;
    register short ii=0;
    unsigned long loc, flag;
    struct packet *pos;

    if (last == NULL)
        loc = 0;
    else
      {
        computehash1(last,loc);
        computehash2(last,loc);
        for (flag = FALSE, pos = ht[loc]; pos != NULL;  pos = pos->next)
            if (strcmp(retkey(&(pos->data)), last) == 0)
              {
                flag = TRUE;
                if (pos->next != NULL)
                    return &(pos->next->data);
              }
        if (flag == FALSE)
            return NULL;
        loc++;
      }
    for ( ;(ht[loc] == NULL) && (loc < TableSize); loc++)
        ;
    if (loc == TableSize)
        return (NULL);
    else
        return &(ht[loc]->data);
}


/* called before the user program exits to free all dynamic memory */
void freehashtable()
  {
    long    i;

    for(i=0; i<TableSize; i++)
        if (ht[i] != NULL)
            deletechain(ht[i]);
    free(ht);
```

Accessing the Math Co-Processor from BASIC

R. P. Haviland

FORWARD

Amiga BASIC is a powerful version of the language, with hundreds of commands. It was well matched to the original Amiga 1000, making virtually all the Amiga features available to program control.

Unfortunately, there is one area where the sponsors of the language version and the machine have not kept the language up to later model machine capability. This is the case for the expanded machines with the 68881 numerical co-processor installed, and for the accelerator boards of the same class. Amiga BASIC ignores the existence of the co-processor, so any performance gain is only the result of clock speed changes. This is in contrast to some of the Amiga Library routines which are provided, which check for the presence to the co-processor chip, and utilize it if present.

Amiga BASIC does provide a way of using such a library, but the process is rather involved. The correct library must be opened, in this case the mathieeedoubbas.library or the mathieeedoubtrans.library. This requires the presence of a bmap file, unfortunately not provided for these libraries; it must be generated.

The necessary functions are then made available with the DECLARE FUNCTIONid LIBRARY command. Thereafter, operation is straightforward, the defined function being used the same way as other BASIC functions.

A method of avoiding these steps has been described. Essentially, a "WEDGE" is installed in BASIC, to intercept commands, examine these for new functions, and route them correctly. This technique was very common in the days of the Commodore C-64. It does have a drawback in that speed is reduced by the added steps of command examination. Unfortunately, the program to make this wedge has not been published by its originator.

One way to avoid the roundabout steps is to program in machine language. Again unfortunately, the available public domain assemblers do not include the routines to assemble co-processor instructions. They are in several of the large "commercial duty" assemblers, but these can be expensive. Several of the C language compilers also allow use of the library functions. These techniques have been well covered in a series of Amazing Computing articles by Predmore, starting in issue V4.3 1989.

In my case, I was not willing convert to C programming, or to completely rewrite a number of BASIC programs in regular use. In fact, I consider C to be a poor language for problem solving programs, because of the excessive detail required, detail which detracts from problem consideration. What I wanted was the ability to use the the math coprocessor within the structure and format of Amiga BASIC.

THE CO-PROCESSOR from BASIC

Amiga Basic does provide a tool to do this. This is the CALL command, which has three forms:

```
CALL subroutine

CALL library function

CALL machine language routine
```

With some maneuvering, either can be used to give access to the co-processor. However the machine language routine is faster, and about as easy to use as the others.

The CALL routine does expect the target routine to be at a known address. While this is contrary to general Amiga programming rules, BASIC does provide a command, VARPTR(variablename) which can do the location operation. By far the easiest use is to set up an integer array whose values are the machine language code. The code must end with a return from subroutine code, RTS or &H4E75.

Care must be taken with this, however. Amiga BASIC changes the location of arrays as new variables are encountered. It is necessary to be absolutely certain that the VARPTR command is not followed by a new variable. If this happens, an error in value can result, but more commonly there is a total crash, with a visit from the GURU. Check carefully before the program is run, or play safe, and place the proper VARPTR command just before the CALL. It is easier but more time consuming to reset all array pointers each time a CALL is made.

The CALL does have a powerful feature. It can be written in the form: CALL routineaddress (1paramater, 2parameter—) where 1parameter is the address of the first

parameter (a named variable), etc. These addresses are passed to the routine on the stack, which allows reading input values to the routine, and returning results. There is enormous flexibility here.

THE CO-PROCESSOR

Turning to the 68881 co-processor, it is designed for high accuracy floating point operation. Whereas normal or single precision floating point is 32 digits, and double precision is 64, the co-processor always uses 80 digits for internal operations. Paramater passing can be single or double precision, with automatic conversion, and can also be a special form with 96 bits, 80 usable. Short 16 bit and long 32 bit integers can be used, as well as a special form for decimal digits.

The co-processor has enormous capability. It includes:

8 internal 80 bit registers

The arithmetic operations, +,-,x,/

A table of constants

A set of functions

Register-register operations

Register-memory operations

Test and decide capability.

These are implemented by adding a number of instructions to the normal ones of the 68000 family. All of these are of the hex form Fxxx, followed by one or more additional words. Because of the structure, these are usually called the F-instructions.

USE OF RPN ARITHMETIC

After several practice runs, it was decided that a reasonable compromise between speed and programming complexity could be had by structuring a set of co-processor accessing machine language routines to emulate a RPN scientific calculator. This is the type introduced by Hewlett-Packard just before computers started their popularity leap. This use has the advantage of previous acquaintance. It is familiar to most users of floating point arithmetic, and in any event is not difficult to learn. For the unintiate, in RPN the common process of addition is written as X Y +, rather than X+Y. This requires a stack to hold "pending operations", but eliminates the need for parenthesis. Internal computer arithmetic is often in RPN format, even though the programming uses the common form.

In keeping with RPN usage, the first four of the 68881 registers are used as a stack, designated as X,Y,Z,T. Since the registers are independent, FMOVE instructions are used as needed, with an extra register used for temporary storage. Operations are set to follow these rules:

The input operation FSET copies the specified variable to the X register, the original value being lost;

The FENTER operation copies X to Y, Y to Z and Z to T: the original value in T is lost;

The FEX operation exchanges X and Y;

The FUP operation copies X to Y, Y to Z, Z to T and T to X;

The FDOWN operation copies T to Z, Z to Y, Y to X and X to T;

The four arithmetic operations, FADD, FSUB, FMUL and FDIV combine X and Y appropriately, and place the result in X: Z is moved to Y and T is copied to Z: The original T remains: note that arithmetic operations in BASIC are unchanged;

The constant operations read the selected constant into X: the original value is lost;

The function operations replace the value in X by the selected function of X. An exception if FSINCOS, which places the cosine in X and the sine in Y: original values of both are lost;

The output operation FGET copies X into the specified variable, and nothing more.

The selected constants were FPI, FZERO (used to clear X), and FBNL (or e, the base of natural logs). Other constants are available.

The selected functions were:

FSQ	FSQR	FABS	FACOS
FASIN	FATAN	FATANH	FCOS
FCOSH	FETOX	FETOXM1	FLOG10
FLOG10	FLOG2	FLOGN	FLOGNP1
FCHS	FSIN	FSINH	FTAN
FTANH	FTENTOX	FTWOTOX	

There is no provision for test and decision, or for such steps as divide-by-zero protection. These must be done in BASIC.

Input and output is double precision floating point. If another format is needed, the conversion must be done by BASIC.

THE MACHINE LANGUAGE ROUTINES

The machine language routines are held in arrays, one for each operation. These are named for the operation. Using FADD as the example, its array is first dimensioned, by DIM FADD%(8). The 9 array elements are then directly read into the array by a series of let statements. The address of FADD is set by FADD&=VARPTR(FADD%(0)). As noted above, this step must be repeated to avoid crashes induced by BASIC moving arrays.

For convenience, the ensemble of machine code is written as a block with BASIC labels in the 64000s. Similarly, the address assignments are a block at 65000. See listings 1 and 2.

The reason numeric labels are used is that the ON ERROR routine of BASIC will only tabulate errors by the nearest preceding label. The sparse label configuration so popular in "Structured Programming" may leave you searching through many lines of code for the error. The simple routines of listing 4 can be added to any program to simplify error finding. (If you are addicted to prettified code with block labels, indenting, etc, just add the line numbers. Once you debug and ensure the accuracy of 2000 or so lines of scientific code, you will appreciate the reason).

If you wish, a set of instructions can be included in the program. Listing 4 is an example.

It is worthwhile to spend a little time becoming familiar with RPN usage. Listing 5 is a core program for this. It first loads the value of PI in X then replaces it with the functional value of 2 to the PI power. The answer is copied to the variable A#.

Use variations of the 500 block to exercise other operations. For example., insert 505 CALL FENTR&, and 515 CALL FADD& to see the result of PI+2 to the PI power. In RPN notaion, this sequence is FPI, FENTER, FTWOTOX, FADD, FGET(A#).

BENCHMARK TESTS

The Savage benchmark is especially suited to a test of co-processor performance. Listing 6 shows a form of this in BASIC, to give a reference of performance without the co-processor.

Listing 7 shows the same program in RPN form. Here, the unused elements of listings 1 and 2 have been deleted, to give as short a program as possible in this form. This step is not necessary, but saves program space and a little run time.

Finally, listing 8 shows an extension of program 7 to increase execution speed. Instead of calling separate machine language routines for each of the successive operations, they are combined into a single special routine, and given a special name. For example, the block from 410 to 480 in listing 7 is replace by the array SAVAGE%. The elements of this array are generated from the individual operations by deleting the RTS instruction &H4E75 from all but the very last operation. The resulting array has a dimension of 19. The advantage of this combination is that the computing overhead associated with switching between BASIC and assembly is greatly reduced.

Running these last four listed programs will give a good picture of the improvement in accuracy and speed provided by the co-processor.

The following was found in a series of tests. All are for 10000 iterations, except the first, which is shown as 10 times the duration for 1000 iterations. The compiler used was AC-BASIC.

Amiga 500
>Listing 6, 434.8 seconds
>Listing 6, compiled, 30.5 seconds

Amiga 1000 plus Sapphire co-processor/accelerator
>Listing 6, 387.1 seconds
>Listing 7, 30.2 seconds
>Listing 8, 10.8 seconds
>Listing 6, compiled, 30.3 seconds
>Listing 7, compiled, 5.72 seconds
>Listing 8, compiled, 1.94 seconds

Amiga 2500, includes co-processor
>Listing 6, 88.4 seconds
>Listing 7, 7.40 seconds
>Listing 6, compiled 57.3 seconds
>Listing 7, compiled, 1.68 seconds.

For some reason, the 2500 showed the poorest accuracy in BASIC, the average error being 7D-5 per operation compiled and 2D-9 not compiled. For the 500 the corresponding values were 9D-9 and 2D-11. The enhanced 1000 gave zero error compiled (less than double precision resolution) and 2D-9 not compiled. With the co-processor in use, errors for all were 1.5D-14.

The benefits of both the compiler and the co-processor are very evident. And, of course, the benefits of the faster clock in the 2500 and the Sapphire board also shows.

EXTENSIONS OF THE TECHNIQUE

There are a number of possible extensions to this technique. One is to use the other three co-processor registers to hold temporary values. Another is to use either main or co-processor registers in index form to control loop execution. There are additional forms of many commands, for example, direct load and add. And the test provisions could be activated, to eliminate the need for this in BASIC.

In applying these techniques, remember the rule that 20 percent of the code uses 80 percent of the time. Use the simplest form initially, then, if more speed is needed, use the form of listing 8 for the time consuming long loops.

Time reductions of 10:1 are easily possible, and about 100:1 can be reached by best use of capability. In summary, a co-processor plus a little effort can be very worthwhile.

```
63999 REM ROUTINE TO FILL RPN ARRAYS
64000 DIM FSET%(4)
64001 FSET%(0)=&H226F:FSET%(1)=&H4:FSET%(2)=&HF211:FSET%(3)=&H5400:FSET%(4)=&H4E75
64010 DIM FGET%(4)
64011 FGET%(0)=&H226F:FGET%(1)=&H4:FGET%(2)=&HF211:FGET%(3)=&H7400:FGET%(4)=&H4E75
64020 REM END OF DATA I/O
64030 DIM FENTR%(6)
64031 FENTR%(0)=&HF200:FENTR%(1)=&H980:FENTR%(2)=&HF200:FENTR%(3)=&H500
64032 FENTR%(4)=&HF200:FENTR%(5)=&H80:FENTR%(6)=&H4E75
64040 DIM FUP%(10)
64041 FUP%(0)=&HF200:FUP%(1)=&HF80:FUP%(2)=&HF200:FUP%(3)=&H980
64042 FUP%(4)=&HF200:FUP%(5)=&H500:FUP%(6)=&HF200:FUP%(7)=&H80
64043 FUP%(8)=&HF200:FUP%(9)=&H1C00:FUP%(10)=&H4E75
64050 DIM FDOWN%(10)
64051 FDOWN%(0)=&HF200:FDOWN%(1)=&H380:FDOWN%(2)=&HF200:FDOWN%(3)=&H400
64052 FDOWN%(4)=&HF200:FDOWN%(5)=&H880:FDOWN%(6)=&HF200:FDOWN%(7)=&HD00
64053 FDOWN%(8)=&HF200:FDOWN%(9)=&H1D80:FDOWN%(10)=&H4E75
64060 DIM FEX%(6)
64061 FEX%(0)=&HF200:FEX%(1)=&H380:FEX%(2)=&HF200:FEX%(3)=&H400
64062 FEX%(4)=&HF200:FEX%(5)=&H880:FEX%(6)=&H4E75
64070 REM END OF STACK MANIPULATION
64080 DIM FADD%(8)
64081 FADD%(0)=&HF200:FADD%(1)=&HA2:FADD%(2)=&HF200:FADD%(3)=&H400
64082 FADD%(4)=&HF200:FADD%(5)=&H880:FADD%(6)=&HF200:FADD%(7)=&HD00
64083 FADD%(8)=&H4E75
64090 DIM FSUB%(8)
64091 FSUB%(0)=&HF200:FSUB%(1)=&HA8:FSUB%(2)=&HF200:FSUB%(3)=&H400
64092 FSUB%(4)=&HF200:FSUB%(5)=&H880:FSUB%(6)=&HF200:FSUB%(7)=&HD00
64093 FSUB%(8)=&H4E75
64100 DIM FMUL%(8)
64101 FMUL%(0)=&HF200:FMUL%(1)=&HA3:FMUL%(2)=&HF200:FMUL%(3)=&H400
64102 FMUL%(4)=&HF200:FMUL%(5)=&H880:FMUL%(6)=&HF200:FMUL%(7)=&HD00
64103 FMUL%(8)=&H4E75
64110 DIM FDIV%(8)
64111 FDIV%(0)=&HF200:FDIV%(1)=&HA0:FDIV%(2)=&HF200:FDIV%(3)=&H400
64112 FDIV%(4)=&HF200:FDIV%(5)=&H880:FDIV%(6)=&HF200:FDIV%(7)=&HD00
64113 FDIV%(8)=&H4E75
64119 REM END OF ARITHMETIC OPERATIONS
64120 DIM FPI%(2)
64121 FPI%(0)=&HF200:FPI%(1)=&H5C00:FPI%(2)=&H4E75
64130 DIM FBNL%(2)
64131 FBNL%(0)=&HF200:FBNL%(1)=&H5C0C:FBNL%(2)=&H4E75
64140 DIM FZERO%(2)
64141 FZERO%(0)=&HF200:FZERO%(1)=&H5C0F:FZERO%(2)=&H4E75
64150 DIM FONE%(2)
64151 FONE%(0)=&HF200:FONE%(1)=&H5C32:FONE%(2)=&H4E75
64159 REM END OF CONSTANTS, START FUNCTIONS
64160 DIM FSQ%(2)
64161 FSQ%(0)=&HF200:FSQ%(1)=&H23:FSQ%(2)=&H4E75
64170 DIM FSQR%(2)
64171 FSQR%(0)=&HF200:FSQR%(1)=&H4:FSQR%(2)=&H4E75
64180 DIM FABS%(2)
64181 FABS%(0)=&HF200:FABS%(1)=&H18:FABS%(2)=&H4E75
64190 DIM FACOS%(2)
64191 FACOS%(0)=&HF200:FACOS%(1)=&H1C:FACOS%(2)=&H4E75
64200 DIM FASIN%(2)
64201 FASIN%(0)=&HF200:FASIN%(1)=&HC:FASIN%(2)=&H4E75
64210 DIM FATAN%(2)
64211 FATAN%(0)=&HF200:FATAN%(1)=&HA:FATAN%(2)=&H4E75
64220 DIM FATANH%(2)
64221 FATANH%(0)=&HF200:FATANH%(1)=&HD:FATANH%(2)=&H4E75
64230 DIM FCOS%(2)
64231 FCOS%(0)=&HF200:FCOS%(1)=&H1D:FCOS%(2)=&H4E75
64240 DIM FCOSH%(2)
64241 FCOSH%(0)=&HF200:FCOSH%(1)=&H19:FCOSH%(2)=&H4E75
64250 DIM FETOX%(2)
64251 FETOX%(0)=&HF200:FETOX%(1)=&H10:FETOX%(2)=&H4E75
64260 DIM FETOXM1%(2)
64261 FETOXM1%(0)=&HF200:FETOXM1%(1)=&H8:FETOXM1%(2)=&H4E75
64270 DIM FLOG10%(2)
64271 FLOG10%(0)=&HF200:FLOG10%(1)=&H15:FLOG10%(2)=&H4E75
64280 DIM FLOG2%(2)
64281 FLOG2%(0)=&HF200:FLOG2%(1)=&H16:FLOG2%(2)=&H4E75
64290 DIM FLOGN%(2)
64291 FLOGN%(0)=&HF200:FLOGN%(1)=&H14:FLOGN%(2)=&H4E75
64300 DIM FLOGNP1%(2)
64301 FLOGNP1%(C)=&HF200:FLOGNP1%(1)=&H6:FLOGNP1%(2)=&H4E75
64310 DIM FCHS%(2)
64311 FCHS%(0)=&HF200:FCHS%(1)=&H1A:FCHS%(2)=&H4E75
64320 DIM FSIN%(2)
64321 FSIN%(0)=&HF200:FSIN%(1)=&HE:FSIN%(2)=&H4E75
64330 DIM FSINCOS%(2)
64331 FSINCOS%(C)=&HF200:FSINCOS%(1)=&H31:FSINCOS%(2)=&H4E75
64340 DIM FSINH%(2)
64341 FSINH%(0)=&HF200:FSINH%(1)=&H2:FSINH%(2)=&H4E75
64350 DIM FTAN%(2)
64351 FTAN%(0)=&HF200:FTAN%(1)=&HF:FTAN%(2)=&H4E75
64360 DIM FTANH%(2)
64361 FTANH%(0)=&HF200:FTANH%(1)=&H9:FTANH%(2)=&H4E75
64370 DIM FTENTOX%(2)
64371 FTENTOX%(0)=&HF200:FTENTOX%(1)=&H12:FTENTOX%(2)=&H4E75
64380 DIM FTWOTOX%(2)
64381 FTWOTOX%(0)=&HF200:FTWOTOX%(1)=&H11:FTWOTOX%(2)=&H4E75
64400 RETURN
```

LISTING TWO

```
64999 REM ROUTINE TO SET ADDRESSES OF RPN ARRAYS
65000 FSET&=VARPTR(FSET%(0))
65010 FGET&=VARPTR(FGET%(0))
65020 REM END OF DATA I/O
65030 FENTR&=VARPTR(FENTR%(0))
65040 FUP&=VARPTR(FUP%(0))
65050 FDOWN&=VARPTR(FDOWN%(0))
65060 FEX&=VARPTR(FEX%(0))
65070 REM END OF STACK MANIPULATION
65080 FADD&=VARPTR(FADD%(0))
65090 FSUB&=VARPTR(FSUB%(0))
65100 FMUL&=VARPTR(FMUL%(0))
65110 FDIV&=VARPTR(FDIV%(0))
65119 REM END OF ARITHMETIC OPERATIONS
65120 FPI&=VARPTR(FPI%(0))
65130 FBNL&=VARPTR(FBNL%(0))
65140 FZERO&=VARPTR(FZERO%(0))
65150 FONE&=VARPTR(FONE%(0))
65159 REM END CONSTANTS, START OPERATIONS
65160 FSQ&=VARPTR(FSQ%(0))
65170 FSQR&=VARPTR(FSQR%(0))
65180 FABS&=VARPTR(FABS%(0))
65190 FACOS&=VARPTR(FACOS%(0))
65200 FASIN&=VARPTR(FASIN%(0))
65210 FATAN&=VARPTR(FATAN%(0))
65220 FATANH&=VARPTR(FATANH%(0))
65230 FCOS&=VARPTR(FCOS%(0))
65240 FCOSH&=VARPTR(FCOSH%(0))
```

```
65250 FETOX&=VARPTR(FETOX%(0))
65260 FETOXM1&=VARPTR(FETOXM1%(0))
65270 FLOG10&=VARPTR(FLOG10%(0))
65280 FLOG2&=VARPTR(FLOG2%(0))
65290 FLOGN&=VARPTR(FLOGN%(0))
65300 FLOGNP1&=VARPTR(FLOGNP1%(0))
65310 FCHS&=VARPTR(FCHS%(0))
65320 FSIN&=VARPTR(FSIN%(0))
65330 FSINCOS&=VARPTR(FSINCOS%(0))
65340 FSINH&=VARPTR(FSINH%(0))
65350 FTAN&=VARPTR(FTAN%(0))
65360 FTANH&=VARPTR(FTANH%(0))
65370 FTENTOX&=VARPTR(FTENTOX%(0))
65380 FTWOTOX&=VARPTR(FTWOTOX%(0))
65390 RETURN
```

LISTING THREE

```
1 REM ERROR SERVER ROUTINES
3 ON ERROR GOTO 60000
60000  PRINT "ERROR ";ERR;" AT LINE ";ERL
60100  REM EXAMPLE OF DIVIDE BY ZERO ROUTINE
60110  IF ERL=X THEN VARIABLE=VARIABLE+1E-09:RESUME
60200  REM EXAMPLE OF A FILE NOT FOUND ROUTINE
60210  IF ERL=Y THEN PRINT "FILE NOT FOUND-REENTER FILEPATH, NAME":  RESUME Z
60300  REM EXAMPLE OF A RETURN TO MENU
60310  IF ERL=P THEN INPUT "PRESS RETURN TO RECOVER FROM ERROR";T$:RESUME Q
60400  REM EXAMPLE OF CLOSEOUT
60410  PRINT "ERROR RECOVERY NOT POSSIBLE":STOP
```

LISTING FOUR

```
63000 'USING THE RPN CALCULATOR FROM BASIC
63010 'WRITE THE EQUATIONS IN RPN FORM; CHECK THAT THEY CAN BE SOLVED
63020 'WITH A 4 REGISTER STACK. USE INTERMEDIATE VARIABLES IF NECESSARY.
63030 'SELECT THE VARIABLE NAMES, APPENDING A # FOR DOUBLE PRECISION.
63040 'TRANSLATE THE RPN EQUATIONS TO COMPUTER FORM. IT IS USUALLY EASIER
63050 'TO PLACE EACH OPERATION CALL ON A SEPARATE LINE.
63060 'ADD THE 64000 AND 65000 ROUTINES AND THE NECESSARY GOSUB COMMANDS.
63070 'WATCH THAT ROUTINE ADDRESSES ARE CORRECTED AFTER VARIABLES ARE
63080 'INTRODUCED. AN INITIAL VARIABLE#=0 CAN SIMPLIFY WORK.
63090 'EXERCISE THE PROGRAM WITH A KNOWN SET OF VALUES.
63100 'WHEN RUNNING CORRECTLY, EVALUATE BENEFIT OF WRITING SPECIAL ROUTINES
63110 'TO ELIMINATE MOVEMENT BETWEEN BASIC AND RPN. THIS SAVES TIME,
63120 'ESPECIALLY WITHIN LOOPS.
63200 'BE GENEROUS WITH BACK-UP COPIES. IT IS EASY TO MAKE A MISTAKE
63210 'WHICH WILL INDUCE A CRASH. AVOIDING THESE IS EASIER WITH PRACTICE.
```

LISTING FIVE

```
10 REM TEST ROUTINE FOR RPN USE OF 68881 TEST/PRACTICE
20 REM 68881 CALCULATIONS IN EXTENDED 80 BIT
30 REM BASIC TO 68881 ALWAYS DOUBLE PRECISION FLOATING POINT
40 REM RPN STACK FOLLOWS RPN X,Y,Z,T RULES
100 A#=0            'NECESSARY TO SET VARIABLES
110 B#=1
200 GOSUB 64000     'FILLS ARRAYS WITH ML ROUTINES
210 GOSUB 65000     'SETS ADDRESSES OF ML ROUTINES
300 APTR&=VARPTR(A#) 'EACH VARIABLE NEEDS A POINTER
310 BPRT&=VARPTR(B#) 'ADD VARIABLES IF NEEDED
400 GOSUB 65000     'CORRECTS ADDRESSES IN NON-COMPILED BASIC
410                 'SINCE ARRAYS ARE MOVED AS VARIABLES ARE CREATED
```

```
420                 'EXPECT ERRORS OR CRASHES IF NOT PROPERLY USED
500 CALL FPI&       'EXAMPLE OF SETTING A CONSTANT?
510 CALL FTWOTOX&   'EXAMPLE OF AN OPERATION
520 CALL FGET&(APTR&) 'EXAMPLE OF I/O
530 CALL FEX&       'EXAMPLE OF STACK MANIPULATION
540 CALL FGET&(BPTR&)
600 PRINT A#        'CHECK RESULTS
610 PRINT B#
700 REM ADD ROUTINES OF LISTING1 AND LISTING2 HERE
710 REM (64000 AND 65000 SERIES)
720 REM THESE ARE NEEDED TO COMPLETE THE PROGRAM
730 REM MODIFY THE 500 BLOCK OF CALLS FOR OTHER OPERATION TYPES
```

LISTING SIX

```
10 REM BASIC SAVAGE BENCHMARK
100 INPUT "ENTER NUMBER OF LOOPS";LOOP%
110 START=TIMER
200 A#=1#
210 FOR N%=1 TO LOOP%
220 A#=TAN(ATN(EXP(LOG(SQR(A#^2)))))+1
230 NEXT N%
300 TEND=TIMER
310 PRINT "RUN DURATION=";TEND-START;"SECONDS
320 PRINT "ERROR=";A#/(LOOP%+1)-1
400 INPUT "PRESS RETURN TO END";T$
410 END
500 REM NOTE EFFECT OF USING A#*A# IN LINE 220
```

LISTING SEVEN

```
10 REM REVERSE POLISH SAVAGE BENCHMARK USING 68881 CALLS FROM BASIC
100 A#=0  'SET VARIABLES, NOT REQUIRED IF SECOND CALL 65000 IS CORRECTLY USED
110 N%=0
120 LOOP%=0
130 START=0
140 APTR&=VARPTR(A#)
200 GOSUB 64010   'FILL ARRAYS WITH ML
210 GOSUB 65010   'SET RPN POINTERS
220 INPUT "ENTER NUMBER OF LOOPS";LOOP%
230 GOSUB 65010
300 START=TIMER
310 CALL FONE&    'FILL STACK WITH ONES
320 CALL FENTR&
330 CALL FENTR&
340 CALL FENTR&
410 FOR N%=1 TO LOOP%
420 CALL FSQ&
430 CALL FSQR&
440 CALL FLOG10&
450 CALL FTENTOX&
460 CALL FATAN&
470 CALL FTAN&
480 CALL FADD&
490 NEXT N%
500 CALL FGET&(APTR&)
600 TEND=TIMER
610 PRINT "RUN DURATION=";TEND-START;"SECONDS
620 PRINT "ERROR=";A#/(LOOP%+1)-1
700 INPUT "PRESS RETURN TO END",T$
1000 STOP
64010 DIM FGET%(4)
64011 FGET%(0)=&H226F:FGET%(1)=&H4:FGET%(2)=&HF211:FGET%(3)=&H7400:FGET%(4)=&H4E75
```

```
64030 DIM FENTR%(6)
64031 FENTR%(0)=&HF200:FENTR%(1)=&H980:FENTR%(2)=&HF200:FENTR%(3)=&H500
64032 FENTR%(4)=&HF200:FENTR%(5)=&H80:FENTR%(6)=&H4E75
64080 DIM FADD%(8)
64081 FADD%(0)=&HF200:FADD%(1)=&HA2:FADD%(2)=&HF200:FADD%(3)=&H400
64082 FADD%(4)=&HF200:FADD%(5)=&H880:FADD%(6)=&HF200:FADD%(7)=&HD00
64083 FADD%(8)=&H4E75
64150 DIM FONE%(2)
64151 FONE%(0)=&HF200:FONE%(1)=&H5C32:FONE%(2)=&H4E75
64160 DIM FSQ%(2)
64161 FSQ%(0)=&HF200:FSQ%(1)=&H23:FSQ%(2)=&H4E75
64170 DIM FSQR%(2)
64171 FSQR%(0)=&HF200:FSQR%(1)=&H4:FSQR%(2)=&H4E75
64210 DIM FATAN%(2)
64211 FATAN%(0)=&HF200:FATAN%(1)=&HA:FATAN%(2)=&H4E75
64270 DIM FLOG10%(2)
64271 FLOG10%(0)=&HF200:FLOG10%(1)=&H15:FLOG10%(2)=&H4E75
64350 DIM FTAN%(2)
64351 FTAN%(0)=&HF200:FTAN%(1)=&HF:FTAN%(2)=&H4E75
64370 DIM FTENTOX%(2)
64371 FTENTOX%(0)=&HF200:FTENTOX%(1)=&H12:FTENTOX%(2)=&H4E75
64400 RETURN
65010 FGET&=VARPTR(FGET%(0))
65030 FENTR&=VARPTR(FENTR%(0))
65080 FADD&=VARPTR(FADD%(0))
65150 FONE&=VARPTR(FONE%(0))
65160 FSQ&=VARPTR(FSQ%(0))
65170 FSQR&=VARPTR(FSQR%(0))
65210 FATAN&=VARPTR(FATAN%(0))
65270 FLOG10&=VARPTR(FLOG10%(0))
65350 FTAN&=VARPTR(FTAN%(0))
65370 FTENTOX&=VARPTR(FTENTOX%(0))
65390 RETURN
```

LISTING EIGHT

```
10 REM REVERSE POLISH SAVAGE BENCHMARK USING 68881 CALLS FROM BASIC
20 REM  AND WITH SAVAGE ROUTINE ENTIRELY IN ASSEMBLY
100 A#=0   'SET VARIABLES
110 N&=0
120 LOOP&=0
130 START=0
140 APTR&=VARPTR(A#)
200 GOSUB 63000   'FILL ARRAYS WITH ML
210 GOSUB 65000   'SET RPN POINTERS
220 INPUT "ENTER NUMBER OF LOOPS";LOOP&
230 GOSUB 65000    'INSURANCE RESET
300 START=TIMER
310 CALL FONE&   'FILL STACK WITH ONES
320 CALL FENTR&
330 CALL FENTR&
340 CALL FENTR&
400 FOR N&=1 TO LOOP&
410 CALL FSAV&(APTR&)
420 NEXT N&
500 CALL FGET&(APTR&)
600 TEND=TIMER
610 PRINT "RUN DURATION=";TEND-START;"SECONDS
620 PRINT "ERROR=";A#/(LOOP&+1)-1
700 INPUT "PRESS RETURN TO END",T$
1000 STOP
63000 'SAVAGE IN ASSEMBLY
63010 DIM FSAV%(19)
```
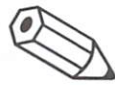
```
63020 FSAV%(0)=&HF200:FSAV%(1)=&H23         'SQ
63040 FSAV%(2)=&HF200:FSAV%(3)=&H4          'ROOT
63050 FSAV%(4)=&HF200:FSAV%(5)=&H15         'LOG10
63060 FSAV%(6)=&HF200:FSAV%(7)=&H12         '10^X
63070 FSAV%(8)=&HF200:FSAV%(9)=&HA          'ATAN
63080 FSAV%(10)=&HF200:FSAV%(11)=&HF        'TAN
64090 FSAV%(0)=&HF200:FSAV%(1)=&HA2:FSAV%(2)=&HF200:FSAV%(3)=&H400   'ADD
64091 FSAV%(4)=&HF200:FSAV%(5)=&H880:FSAV%(6)=&HF200:FSAV%(7)=&HD00
63100 FSAV%(12)=&H4E75                      'RTS
64010 DIM FGET%(4)
64011 FGET%(0)=&H226F:FGET%(1)=&H4:FGET%(2)=&HF211:FGET%(3)=&H7400:FGET%(4)=&H4E75
64030 DIM FENTR%(6)
64031 FENTR%(0)=&HF200:FENTR%(1)=&H980:FENTR%(2)=&HF200:FENTR%(3)=&H500
64032 FENTR%(4)=&HF200:FENTR%(5)=&H80:FENTR%(6)=&H4E75
64150 DIM FONE%(2)
64151 FONE%(0)=&HF200:FONE%(1)=&H5C32:FONE%(2)=&H4E75
64400 RETURN
65000 FSAV&=VARPTR(FSAV%(0))
65010 FGET&=VARPTR(FGET%(0))
65030 FENTR&=VARPTR(FENTR%(0))
65150 FONE&=VARPTR(FONE%(0))
65390 RETURN
```

These worldwide Amazing Dealers carry Amazing Computing, your best resource for information on the Amiga™, and AC's GUIDE, the only complete Amiga product guide. To become an Amazing Dealer, please call: **1-800-345-3360**

ALASKA
•Best Electronics — Anchorage
Juneau Electronics — Juneau

ALABAMA
•Alabama Computers Inc — Huntsville
Little Professor Books — Homewood
Madison Book & Computer — Madison
Paeco Industries — Birmingham
•The Computer Image — Birmingham
The Computer Nut — Birmingham
Universal Computer System — Mobile

ARKANSAS
Commodore Connection — North Little Rock
Commodore Connection — Sherwood
Computer Concepts — Gosnell
Microtronix — Fort Smith
The Micro Shop — Little Rock

ARIZONA
Bookman's — Tucson
Books Brothers Ltd. — Tucson
Books Etc. — Tempe
Bookstar — Phoenix
Bookstar — Scottsdale
Brain Child Software — Phoenix
Computer Libray, Inc. — Tempe
Computer Works — Glendale
•Computer Y's — Tucson
Copperstate Business — Phoenix
Livingston's Books, Inc. — Sierra Vista
North American Digital — Tucson
Try Tech Inc. — Phoenix

CALIFORNIA
3 E Software And Systems — Hayward
A-1 Software Express — Lancaster
Aday International — Foster City
American Mags & Books Ltd. — Englewood
Arcadia Newsstand — Arcadia
B. Dalton Books — San Rafael
Balboa Office Supply — San Diego
Bits & Bytes — Fairfield
Bits n Pieces Comp Prod. — Anaheim
Bookland — Marysville
Bookstar — Brea
Bookstar — San Diego
Brentano's — Sacramento
Brentano's — San Francisco
Brown Knows Computing — Redlands
Bungalo News — Pasadena
CAL-Tech Bookstore — Pasadena
Candy Computer — Elk Grove
•Capri Products Company — Burlingame
Castro Kiosk — San Francisco
Catri Product Company — Burlingame
Centerfold News — San Mateo
Central Park Bookstore — San Mateo
Century Computer Systems — La Habra
Circus of Books — Los Angeles
Circus of Books — Los Angeles
•Commodore Land — Alhambra
Commtech — National City
Computer Attic — Palo Alto
Computer Depot — San Marcos
Computer Literacy Books — Sunnyvale
Computer Literacy — San Jose
Computer Literacy — Santa Clara
•Computer Nook, Inc. — San Barnadino
Computer Outlet — Chula Vista
Computer Outlet — San Marco
Computer Room — Scotts Valley
Computer Showcase — San Francisco
Computer Spectrum — Burlingame
Computer Time — Citrus Hts
Computers Plus — Grove City
Coronet Newsstand — Oceanside
Covina Bookstore — Covina
Dash Software — Auburn
•Dataphile — Watsonville
Dave's Smoke Shop — Berkeley
Del Amo Books & News — Torrance
Delauriers News Agency — Oakland
Delta Software — Antioch
Discount Computer Station Inc. — Long Beach
ECX Computer Co. — Walnut Creek
Empire Computers — Santa Rosa
Encinitas News — Encinitas
Fil-American — San Francisco
Fisher Computers + Software — Rancho Cucamonga
•Harding Way News — Stockton
Hoenicchson, Schneider & Young, Inc. — Berkeley
Home Computing Center — Concord
Home Computing Center — Mill Valley
Home Computing Center — San Leandro
HT Enterprises — Sunnyvale
Int'l Periodical Dist — San Diego
International Telecine — Santa Ana
Jech Distributing — San Diego
•K J Computers — Granada Hills
Kiosk Plus — San Francisco
L.A. Int'l News — Redondo Beach
L. A. Int'l News — Venice
Leed & Wood Co. Ltd. — Englewood
Little Professor Books — Placentia
Lyon Enterprises — Berkeley
Lyon Enterprises — Berkeley
•Micro Galaxy — Santa Rosa
Micro P. Technologies — Lawndale
Mission News — San Luis Obispo
NewsBeat — Davis
Norma McCoy — San Diego
Northridge Newsstand — Northridge
Op Amp Tech Books — Los Angeles
Paras Newsstand — San Diego
Parsin International — San Diego
Pop's Int'l News — Palo Alto
Printers Inc. — Palo Alto
Put's Electronics — North Highlands
Ridgecrest Computer Ctr. Inc. — Ridgecrest
Robertson Magazine & Book — Los Angeles
S.O.S Computers — Los Angeles
Sawyer's News Inc. — Santa Ana
SeventhNearB — San Diego
Silver Strand Computers — Imperial Beach
Simi Valley Newsstand — Simi Valley
Sinkyou Business Systems — Mount Shasta
Solmap Products U.S.A. — Englewood
Software Cellar — Anaheim
Software Central — Pasadena
Software First — Santa Rosa
Software First — Vallejo
Software Industry & General Hardware — Pomona
Software Plus — Citrus Heights
Software Plus — El Cerrito
Software Plus Plus — Modesto
Stacy's #5 — South Pasadena
Technitronics Service — San Francisco
The Booksmith — Tracy
The Computer Connection — El Cajone
•The Lively Computer — Chula Vista
Third Avenue News — Chico
Tower Books — Citrus Hts.
Tower Books — Concord
Tower Books (3 locations) — Sacramento
Tower Books — San Mateo
Tower Books — West Sacramento
Universal News — Hollywood
Vintage Wine — San Diego
WCI Computer Central — Eureka
Warner Engineering — San Diego
Winners Circle — Berkeley
Woodland Hills — Woodland Hills
World Book & News — Hollywood

COLORADO
Amazing Computers — Denver
Aurora Newsstand — Aurora
Citadel Computers — Colorado

Springs
•Computer Discount — Denver
Computer Repeats — Boulder
Computer Room — Aurora
Main Street Software — Colorado Springs
Micro World — Lakewood
Paragon Computers — Colorado Springs
Paragon Software & Electronics — Colorado Springs
Southwest Newsland — Littleton
Sun Country Computers — Colorado Springs
Tattered Cover Bookstore — Denver
Westminister Newsland — Westminister

CONNECTICUT
Aetna Life Club Store — Hartford
Beta Press — Fairfield
Computers' Etc. — Fairfield
Derric Electronics — Hamden
Friendly Computers — Southington
•Infotronics, Inc. — Woodbury
Lac Distributors — Norwalk
Lebanon/Levant Dist. — Derby
Liberty Book — Derby
Mnemonics Inc. — Norwalk
One Byte — Quaker Hill
Softwar Inc. — Middlebury
Software Kingdom — East Hartford
Tricom Computers — North Windham

DISTRICT OF COLUMBIA
Programs — Washington
World News — Washington

DELAWARE
Newark Newsstand — Newark

FLORIDA
Amazing Computers. — Orlando
•Ami Comp Computer Center — Oviedo
Amoury Company, Ltd. — Miami
Atlantic News — Daytona Beach
Bookstop — Altamont
Bookstop — Clearwater
Bookstop — Miami
Bookstop — N. Palm Beach
Brentano's — Vera Beach
Chuck's Newsstand — Melbourne
City Newsstand — Pinellas Park
Commodore Country Inc. — Pennsacola
•Computer Bar — Miami
•Computer Image — Miami
Computer Lane Inc — Pembroke Pines
Computer Room — Ft. Lauderdale
Computer Specialties — Melbourne
Computers + Video Creations — Palm Beach
Computers Etc. — Sarasota
•Computers Plus — S. Daytona
Creative Equipment — Miami
Design Computer Systems Inc. — Lake Worth
Extra Extra — Tampa
Florida Book Store — Gainesville
Gen-Reba — Gainesville
Gulf Coast Computer Exchange — PanamaCity
Joe's News, Inc. — Miami
Malvar Freight — Miami
Mark Twain Smoke Shop — Lakeland
Micro Computer City — Lansing
New Age Electronics — Clearwater
New Age Electronics — Largo
Orange Avenue News — Orlando
PC Collection — Coral Gabels
Program Store — Clearwater
Program Store — Pinellas Park
Prospect News & Books — Ft. Lauderdale
Public Domain Solutions — Sarasota
Software City — Orange Park
Sunny's At Sunrise, Inc. — Sunrise
Sunshine Software — Ft. Lauderdale
Universal Books, Inc. — Kissimmee
Worldwide News — North Miami Beach

GEORGIA
Ampex System — Norcross
BFJ Enterprises — Roswell
Bookstar — Marietta
Brentano's — Atlanta
Cardinal Computers — Dalton
Elter News Center — Forest Park
Jackson's Computer Center — Savannah
Micro Center/Marietta — Marietta
Mostly Software — Columbus
Oxford at Buckhead — Atlanta
Oxford Books, Inc. — Atlanta
Reader's Market — Valdosta
Software City Savannah — Savannah
•Software South Inc. — St. Mary's
•Techno World — Atlanta
The 64 Store — Atlanta

HAWAII
Computronics — Honolulu
Data-One Computer Center — Honolulu
•J & C Computers — Aiea

IDAHO
•A.B.I. Computer & Video — Nampa
Commodore Computer Center — Boise
The Creative Edge — East Ketchum

ILLINOIS
A Byte Better — Loves Park
Audio Visual Req. — Chicago
Books & Bytes, Inc. — Naperville
Computer Software Connection — Naperville
Coopersmith — Shaumburg
Coopersmith — Vernon Hills
Digital World — Naperville
DigitalWorld — Addison
DOS Computer Center — Niles
•FO Software & Cadds — Bloomingdale
Keep In Pace Computers — Champaign
Kroch's & Brentano's — Aurora
Kroch's & Brentano's — Chicago
Kroch's & Brentano's — Orland Park
Kroch's & Brentano's — Schaumburg
M2J2 Inc. — Champaign
•Megabyte Computer Works — Springfield
Memory Expansion — Bloomington
Micro Age — Oaklawn
•Micro Ent Enterprises Inc. — Chicago
Micro Tek Soolutions — Aurora
•Micro Tek — Kankakee
•Ring Computers — Geneva
Software City — Bellivile
•Software Plus Chicago — Chicago
•Software Plus West — Hanover Park
Software Plus — Wheeling
Unique Computer — Sterling
Warren Radio/FRANCO — Peoria
Wizard's Computer Shop — Crystal Lake

INDIANA
Bookland — Indianapolis
Burkat Computer Ctr. — South Bend
•CPU Inc. — Mountain View
Cal Com Inc. — Sacramento
Computer Corner — Kokomo
Computer Corner — Ft. Wayne
Computer People Inc. — Marion
•Computer Room — Michigan City
Computer Room — Merrillville
Digital Arts — Bloomington
Indiana News — Indianapolis
Micro Computer Centers Inc. — Indianapolis
Mid-Kansas Microcomputer — Newton
The Book Corner — Bloomington
•Vons Computers — West Lafayette
•Xerox Computer Center — Indianapolis

IOWA

Computer Solutions & Facts — Des Moines
Computers & More — Des Moine
Gronert Computers — Des Moine
Soft Search — Ames

KANSAS
Arc Electronic Distributors — Wichita
Brentanos — Mission
Micro Media — Hays
Midwest Computers — Manhattan
•Program Store — Princeton
Software City — Overland Park
Software Exchange of KS — Wichita
Thoroughbred Systems — Topeka

KENTUCKY
Computer Dimensions Inc. — Lexington
Computer Mania Inc. — Florence
Expert Services — Florence
Hawley Cookie — Louisville
Mr. Horan's Computer Lab — Louisville
Reality World Computers — Lexington
Softech Computer Systems — Lexington
The Computer Lab — Louisville

LOUISIANA
Bookstar — New Orleans
City Newsstand — Baton Rouge
Lakeside News — Metairie
Modern Business Machines — Metairie
Software Source — Lafayette
The Book Shelf — Slidell

MAINE
Skylight Software — Belfast

MARYLAND
Bowes Books — Lexington Park
Buried Treasure — Rockville
Capital Classics Ltd — Silver Spring
Greetings and Readings — TowsonSystems
Plus — Overland Park
Knowledgeware Systems — Kensington
Micro Computer Center — Baltimore
New Age Computers — College Park
News 2 U — Gaithersburg
Professional Micro Services — Baltimore
Software Advantage — Rockville
Software City — Rockville
Systems Plus — Overland Park

MASSACHUSETTS
•Computer Works — Middletown
Computer Zone — North Attleboro
•LCA Video & Computer Ctr. — Norwood
•The Memory Location — Wellsley
Newstbreak, Inc. — Swansea
OmniTek Computers — Tewksbury
Pioneer Valley Data Equipment — Amherst
Soft Designs — Belmont
Softpro — Burlington
Software Connections — Swansea
The Bit Bucket — West Newton
•Tycom Inc. — Pittsfield
Valdicara — Hadley

MICHIGAN
Barnes & Noble — Rochester Hills
•Better Mousetrap Computers — Flint
•Book Center Of E. Detroit — E. Detroit
Book Stop — Allen Park
Ops — Niles
Borders Book Shop — Okemos
Central Park Computers — Lansing
Community Newscenter — Lansing
CompuSoft — Kalamazoo
Computer Emporium — Holland
Computers And More — Livonia
•Direct Access — Dearborn
Global Computer — Flint
•Kitchen Book — Brighton
Little Professor Books — Johnson City
Michigan News Agency — Kalamazoo
•Michigan Software — Novi
Micro World — Livonia
Micro World — Milford
Micro World — Roseville
Micro World — Southfield
Micro World — Southgate
Micro World — Sterling Heights
Micro World Computer Center — Southfield
Microworld Computer Center — Dearborn
Midland Computer Shop — Midland
Midwest Micro Computer — Lapeer
•Norman Business Control Sys — Wyoming
RACC Enterprises — Alanson
Readmore Newscenter — Lansing
•Rite Way Computers — Warren
Rochester Book Center — Rochester Hills
Slip Disk — Lansing
Slip Disk — Livonia
•Slip Disk — Madison Heights
•Slip Disk — New Baltimore
Slip Disk — Southfield
Slip Disk — St. Clair Shore
Slip Disk — Grand Rapids
Software City — Kalamazoo
•Software House — Minot
T.J. Enterprises — Clinton
The Edge Connector — Clio
•Ye Ole Computer Shoppe — Ypsilanti

MINNESOTA
Brentano's — Minneapolis
Computer Stuff — Minneapolis
Computers, Etc. Inc. — Burnsville
•LNH Software — Maple Grove
Look Ahead Software — Northfield
Shinder's — Bloomington
Shinder's — Burnsville
Shinder's — Crystal
Shinder's — Eden Prairie
Shinder's — Edina
Shinder's — Inver Grove
Hgts — Maplewood
Shinder's — Minneapolis
Shinder's — Minnetonka
Shinder's — Roseville
Shinder's — St. Paul
Shinders — Baline
Shinder's Read More — Minneapolis
•Specialists Inc. — Hopkins
•Valiant Inc. — Stillwater
Wizard's Computer Shop — Crystal Lake

MISSISSIPPI
National Computer Center — Ocean Springs

MISSOURI
Assoc Computer Services — Springfield
Brands Mart Computers — Kansas City
Clark Worldwide — Kansas City
Data Byte Computer Co. — St. Louis
Software City — Independence
Systems Plus of St. Louis — St. Louis
Systems Plus — Overland Park

MONTANA
Compur House — Missoula

NEBRASKA
•Computers "R" Us — Lincoln
•Double E Computer Systems — Omaha
My Com Inc. — Omaha

NEVADA
•Computer House — Sparks
Infinity Computer — Las Vegas
Nat'l Discount Computer — Las Vegas
Computer Room — Las Vegas

NEW HAMPSHIRE
Dartmouth Bookstore — Hanover
•Digital Connection Computers — Dover
Diversified Computers — Keene
System Eyes — Nashua

NEW JERSEY
Act One — Vineland
Atlantic News — Atlantic City

Brentano's — Hackensack
Business System Professional — Scotch Plains
Enachpaa Enterprises — Fort Lee
Family Computer Centres — South Orange
•Garden State News — Cliffside Park
•Mantana, Inc. — Eatontown
McAleavy's Newsstand — Somerville
Program Store — Princeton
Software Station — Rockaway
The Program Store — Eaton Town
The Program Store — Wayne
The Soft Side — Northfield
Unlimited Software Inc. — Rutherford
•Village Computer Center — Cedar Knolls
Waldensoftware — Paramus
Wilson Group USA — Jersey City

NEW MEXICO
Cafe Alegro — Los Alamos
New Horizon Bookstore — Alamogordo
Newsland Bookstore — Albuquerque
•Page One Newsstand — Albuquerque
Road Runner — Santa Fe

NEW YORK
•Amagination — New York
Amcorp Ltd — Huntington
Station — Wappingers Falls
Amicus-Hudson Valley — East Northport
Amigo Business Computers — Huntington
B.C.Communications — Huntington
Station — Huntington
Book Distributor — Huntington
Brentano's — Syracuse
Broadway News — Saratoga
Springs — Buffalo
Buffalo Books Inc. — Merrick
Byte Shop — White Plains
Compurite Center — New Paltz
Computer Metropolis — Jamestown
Computer Outlet — Latham
Contemporary Computing — Binghamton
•Creative Computers — Brooklyn
Cromland Inc. — Brooklyn
Delmar/Buffalo Books — New Hyde Park
•Global Software — New York
Hudson News — New York
Industrial Color Labs — Fayetteville
International Smoke — New York
Island Distributor Computer Services — Coran
Kemco Publisher Ser — East Northport
•Leigh's Computers — New York
Lift Bridge Book Shop — Brockport
MicroWorks — Kenmore
Mnemonics — North Port
Oxide Industries Inc. — Westbury
•Peavey's Enterprises — Latham
GB Software, Inc. — Forest Hills
•Ray Supply Inc. — Plattsburg
Readers Market — Tonawanda
Rochester Books — Rochester
Sherry Tunkel — New York
Singapore Book — Huntington
Station —
Software & Such — Scotia
Software City — Spring Valley
Software City — Fairport
Software Link — White Plains
Software Trading — New York
Syspul Computers — Brooklyn
•The Computer Cellar — Albany
•The WorkWorks — Tonawanda
Tower Books — New York
Ubiquity — Flint
Unicorn Electronics — Johnson City
Video Computer Bin — Brooklyn
•Video Computer Center — Rome
Village Computers — New York
Wild West News — Rochester
Worldwide Media Service — New York

NORTH CAROLINA
Digitz — Raleigh
K&S Newsstand — Clemmons
K&S Newsstand — Kernersville
K&S Newsstand — Winston-Salem
News Center — Cary
Parker News, Inc. — High Point
Serachronic Technologie — Burlington
Software City — Fayetteville
Software City — Raleigh
Software Haus — Charleston

NORTH DAKOTA
Computer Associates — Fargo
G. F. Computers — Grand Forks
The Computer Store — Minot

OHIO
B5 Software — Columbus
Borders Bookshop — Columbus
Brentanos — Cincinnati
Cincinnati Fin SQ News — Cincinnati
Computer Network — Newark
Eddy's Computer Store — Elyria
•Fairborn Home Computer — Dayton
•Infinity Computing — Columbus
Lake's Consumer Electronics — Akron
Lakewood International News — Lakewood
Leo's Book & Wine Shop — Toledo
Little Professor Books — Seattle
Little Professor Books — Columbus
Local Computer Store — Chesterland
Medical Computers — Centerville
•Merical Computers — Centerville
Micro Center-Columbus — Columbus
Micro Center-Westerville — Westerville
Microtech Computer Center Inc. — Toledo
Microwave Magic — Fairfield
News Shop — Baline
•North Coast Programming — Willoughby
Not Just Software — South Toledo
Not Just Software — West Toledo
Paper & Ink — Tiffin
PC Powerhouse, The — Toledo
Power Computing Inc. — Toledo
Salem Computer Center — Salem
Software Centre International — North Olmsted
Software and More — Cincinnati
Software City — Kansas City
Software Connection — Mayfield Hts.
Somers Photo & Comp — Cleveland
Heights —
Upgrade — North Olmsted

OKLAHOMA
Amazing Computer Systems — Oklahoma City
Info 1 Computers — Oaklahoma City
Little Professor Books — Stillwater
Merit Computers — Oklahoma City
Micro Link Computer Store — Tulsa
•Second Hand Software — Oklahoma City
Software City — Midwest City
Video-Comp Inc. — Lawton

OREGON
Andromeda — LaGrande
Clackamas Computers — Beaverton
Clackamas Computers — Clackamas
Comm-Shack — Salem
Libra Books Inc. — Eugene
Software Express — Eugene
News & Smoke Shop MF — Medford
PC Express — Eugene
Peterson's on Morrison — Portland

Powell's Tech Bookstore — Portland
Rich's Cigar Store — Portland
Sixth & Washington News — Portland
So't Cell Software — Portland
Software Pipeline-Salem — Salem
Software Plus — Beaverton
Tech Books — Portland

PENNSYLVANIA
•ATD Software — Sayre
Bryn Mawr News — Bryn Mawr
Business & Computer Books — Willow Grove
CDR Systems — Pittsburgh
Computer Basics Inc. — Erie
Computer Garden — Edwardsville
•Circle Computer — Ephrata
Clark Worldwide — Middletown
•Computer Basics Inc. — Hermitage
Computer Wurx — Stroudsburg
Computers and Games — Reading
•Electronic Connection — Reading
Elmo's Computers — Brownsville
Floppy House Software — Palmyra
Gene's Books — King of Prussia
•Laser-Tosh Computer Solutions — Newtown
Micro Master Inc. — Monaca
•North Central Digital Systems — Danville
•Pittsburgh Computer Store — Pittsburgh
Quinn's MicroAge Center — Moosic
Reader's Market — Erie
•Software Hut Inc — Philadelphia
•Some Hole In The Wall — Philadelphia
T A Slick & Co, Inc — Lancaster
Tasco Computer Center — Lancaster
The Floppy Disk — Mechanicsburg
Triangle Computers — Indiana
Walden Mill Images — Berlin

RHODE ISLAND
•Micro Limits — Warwick
•Software Connections — Warwick
•The Computer Place — Woonsocket

SOUTH CAROLINA
•Axcom International Inc. — West Columbia
Software Haus — Charleston
•Global Software — New York
Watson's Jewelery — Chesterfield

TENNESSEE
Amazing Computers — Goodlettsville
Bristol Micro Technolog — Bristol
ComputerLab of Memphis — Memphis
Compu-Systems Inc. — Nashville
Davis-Kidd Booksellers — Nashville
T.Y. Corp — Bartlett

TEXAS
A&E Computers — Grove
Alamo News & Computer Inc. — San Antonio
All Systems Go Software — San Antonio
Amazing Computer Systems — Dallas
Amazing Computer Systems — Fort Worth
Amazing Computer's — San Antonio
Bookstop — Arlington
Bookstop — Austin
Bookstop (s locations) — Dallas
Bookstop — Corpus Christi
Bookstop — Houston
Bookstop — Mesquite
Bookstop — Plano
Bookstop — San Antonio
Brentano's — Dallas
Computer Ease — Corpus Christi
Computer Magic — Austin
Computer Skills — Eules
FNS Computer Services — Texarkana
Floppy Joe's — College Station
Matagorda Computers — Bay City
Maxwell Book Inc. — Desota
Metropolitan Computer — Richardson
Micro Search — Houston
Software Library — Wichita Falls
Software Terminal — Fort Worth
•The Computer Experience — San Antonio
The Computer Stop! — Amarillo
The Federated Group — Mosquite
Threshold — Amarillo
Trinity News Computer — Fort Worth
University Co-op Society — Austin

UTAH
Armadillo Bros — Provost
Armadillo Bros — Salt Lake City
Magazines and Gifts — Salt Lake City
People's Computers — Provo

VIRGINIA
Audio Fidelity Corp — Richmond
Compsec — Alexandria
•Discovery — Falls Church
Family Computer Center — Fairfax
Family Video & Computer Center — Fairfax
Newport News —
I O Computer — Norfolk
Metro Video & Electronics — Arlington
Tract Software — Princa William
•University Bookstore—VA Tech — Blacksburg
Video Computer Retailers — Chesapeak
•Virginia Micro Systems — Woodbridge

WASHINGTON
68.000 Club — Spokane
Alti News — Seattle
Amazing Electronics — Spokane
Apple Valley Computers — E. Wenatchee
Bits, Bytes and Nibbles — Spokane
Brentano's — Seattle
Bull Dog News — Seattle
•Butlers Computer Services — Federal Way
Byte World — Kent
Capitol News & Mag — Olympia
Computer Concepts — Bothell
Family Computers — Everett
Micro Age Computer Store — Bellevue
•Nybbles & Bytes — Tacoma
•Omni International — Seattle
Port Book and News — Port Angeles
Software Gallery — Walla Walla
Spectral Electronics & Computers — Bellevue
Steve's Broadway News — Seattle
Swan Computers — Yakima
Take A Byte — Kennewick
Tech Star — Kent
The Computer Shoppe — Vancouver
Tower Books — Bellevue
Tower Books — Tacoma
Tower Books — Blaine
Unihall — Federal Way
Y Not Magazines and Books — Federal Way

WEST VIRGINIA
•SDA Computers Inc. — Hurricane

WISCONSIN
20th Century Books — Madison
•Computer Software Center — Milwaukee
•Fox Valley Personal Computer — Oshkosh
Madison Programmer's Guide — Madison
Software City — Wausau
•TMW Software Inc. — Wausau

WYOMING
Computer Works — Laramie
Crazy Mountain Electronics — Casper

ADDITIONAL U.S. DISTRIBUTION BY
Ingram Periodicals Inc. — La Verne, CA
International Periodical Distributors — San Diego, CA

DISTRIBUTION OUTSIDE OF THE U.S.
CANADA
ALBERTA
Coles, Book People — Calgary
Coles, Book People — Edmonton
Software Supermart — Edmonton.

BRITISH COLUMBIA
Compuime Computer & Software — Prince George
Strider Computer Centres — Delta
Unihall — Vancouver

MANITOBA
Adventure Software, Ltd. — Winnepeg

NEW BRUNSWICK
Computer World — Moncton
P.G. Compusystems Inc. —

NEW FOUNDLAND
J. W. Allan Co. Ltd — St.
John's

NOVA SCOTIA
Atlantic News — Halifax
Coles, Book People — Yarmouth
Kobelek Systems — Halifax
Renmart Enter Ltd — Green-
wood

ONTARIO
All Computers — Gloucester
Arkon Electronics Ltd. — Toronto
Book Company — Ottawa
Brittons Smoke Shop — Ottawa
Canadian Software Source — Downsview
Chatham Computer Sales — Chatham
Classic Computer — London
County's News Depot — St.
Thomas
Coles Bookstore — Rexdale
Coles, Book People — Cobourg
Coles, Book People — Gueiph
Scarborough
Coles, Book People — Toronto
Willowdale
Compul-or Software — Barrie
Computer & You, The — Islington
Computer Express — Mississauga
Computer Force — Sudbury
Computer Nook — Chatham
Computer Odyssey, Inc. — Hamilton
ComputerLand — Cornwall
Computronks — Brampton
Computronics Data Centre — Toronto
Compsec — Oshawa
Compute — Oshawa
Cryston Computer System — Midland
CVF Data Systems —
Daily Planet, The — Whitby
Data Marketing Kingston — Kingston
Desktop Computer, Inc. — Kitchener
Electronics 2001 —
Willowdale
Foreign Correspondence — Kingston
Globe Magazine — Ottawa
Gran Bookstore, The — Toronto
Granger T.V. Ltd. — Goderich
JC-Land News — Barrie
Horizon Computers — Whitby
Humber College Stores — Rexdale
Kahle Electronics — Kanata
Lichtman's — Thornhill
Lichtman's (5 locations) — Toronto
Lichtman's —
Willowdale
Lightman's News & Books — Toronto
Lynn Technical Services — Ottawa
Mags & Fags — Ottawa
Mikadon Computer Engineering — Thunder
Bay
Mr. Diskettes — Ottawa
Novatech — Niagra
Falls
Nu-Info Tech — Ottawa
Peach Micro — London
Personal Computers — London
Personal Keyboard Systems — London
Prospero Books Ltd — Etobicoke
Public Book Warehouse — Pickering
Smoker's Corner — Colborne
Smoker's Place — St.
Catherines
The Software Project — Hamilton
U of W Bookstore — Waterloo
World's Biggest 200 — Toronto
Zoez Software —
Brantford
Zoez Software — Milton

QUEBEC
Azimut Infor. Inc — Quebec
Cameoint-info — Montreal
Coles, Book People (2 locations) — Montreal
E. W. Press & Books, Inc. — Montreal,
Jupiter Technologies Inc. — Ft-Claire
Kirk Imports, Reg'd — Sainte-
Foy
Logiciel International — Montreal
Madison Du Logiciel — Montreal
Multimicro — Ville
D'Anjou
Ordinateurs 2000 Inc. — Hull
Quebits Logiciel Inc. — Montreal
Star Tek Infosolutions — St.
Leonard

SASKATEWAN
CD's Computer Connection — Saska-
toon
Computer Answers — Prince
Albert

DISTRIBUTED IN CANADA BY
Ingram Software Distributors — Concord
Compuit Distributors — British
Columbia

DISTRIBUTED IN EUROPE BY
Stars & Stripes

ARGENTINA
Logic Computer — Buenos
Aires

AUSTRALIA
Computermate — Mt.
Kuring-GAI, SW
Diskworks — Green-
wich, NSW
John Verheoeven —
Cloverdale, WA
Run Soft — Melbourne

SAUDI ARABIA
Abdulla Fouod & Sons — Daman

SCOTLAND
•Amiga Center of Scotland — Edinburgh

SWEDEN
Procomp

● **These Amazing Dealers also carry AC's TECH, the #1 disk-based, all-technical publication for Amiga users.**
Amazing Computing is also available in most B. Dalton Booksellers, B. Dalton Software Stores, Crown Books, Software Etc., and selected WaldenBooks Stores and Walden's Software Store locations.

AC's TECH /AMIGA Reader Service Card

Name _____

Street _____

City _____ State ____ ZIP _____

Country _____

A. At your option, please provide the following data:
- O 1. Male
- O 2. Female
- O 3. Married
- O 4. Single/never married
- O 5. Separated/widowed

B. What is your age?
- O 6. Under 18
- O 7. 18–24
- O 8. 25–34
- O 9. 35–49
- O10. 50–64
- O11. 65 or over

C. Which of the following do you now own?
(please check all that apply)
- O12. Amiga 3000
- O13. Amiga 2500
- O14. Amiga 2000
- O15. Amiga 1000
- O16. Amiga 500
- O17. do not own an Amiga

D. Where do you use your Amiga(s), and about how many hours per week do you use an Amiga at each location?
- O18. At home _____ hours per week
- O19. Home office _____ hours per week
  (type of business _____)
- O20. At work _____ hours per week
  (type of business _____)
- O21. At school _____ hours per week
  (applications _____)
- TOTAL: _____ hours per week

E. Please indicate the primary and secondary applications for which you use your Amiga(s):
- 22. Primary _____
- 23. Secondary _____

F. Please indicate the level at which you now consider yourself to be programming the Amiga:
- O24. Beginner
- O25. Intermediate
- O26. Advanced
- O27. Do not program

G. What language do you most often program in?
- 28. _____

H. How much money are you likely to spend on all Amiga product purchases this year? (all personal and business/education for which you have final decision, combined)
- O29. Less than $250
- O30. $251–$500
- O31. $501–$1000
- O32. $1001–$1500
- O33. $1501–$2500
- O34. $2501–$5000
- O35. $5001–$7500
- O36. $7501–$10000
- O37. Over $10000

I. How did you obtain this issue of AC's TECH?
- O38. Preordered this issue prior to its publication.
- O39. Ordered this issue after seeing it elsewhere.
- O40. Ordered a charter subscription.
- O41. Purchased it at my Amiga dealer.

J. How many others not including yourself have read/will read this issue of AC's TECH?
- O42. _____ others, in addition to myself

K. Overall, how would you rate this issue of AC's TECH in the areas indicated?

Editorial content:
- O43. Excellent
- O44. Good
- O45. Fair
- O46. Poor

Disk:
- O47. Excellent
- O48. Good
- O49. Fair
- O50. Poor

L. Do you regularly read or subscribe to Amazing Computing?
- O51. I am a subscriber to AC.
- O52. I read, but do not subscribe to AC.
- O53. Do not read or subscribe to AC.

M. Have you ever purchased a copy of AC's Guide?
- O54. Yes
- O55. No

302

101	102	103	104	105	221	222	223	224	225
106	107	108	109	110	226	227	228	229	230
111	112	113	114	115	231	232	233	234	235
116	117	118	119	120	236	237	238	239	240
121	122	123	124	125	241	242	243	244	245
126	127	128	129	130	246	247	248	249	250
131	132	133	134	135	251	252	253	254	255
136	137	138	139	140	256	257	258	259	260
141	142	143	144	145	261	262	263	264	265
146	147	148	149	150	266	267	268	269	270
151	152	153	154	155	271	272	273	274	275
156	157	158	159	160	276	277	278	279	280
161	162	163	164	165	281	282	283	284	285
166	167	168	169	170	286	287	288	289	290
171	172	173	174	175	291	292	293	294	295
176	177	178	179	180	296	297	298	299	300
181	182	183	184	185	301	302	303	304	305
186	187	188	189	190	306	307	308	309	310
191	192	193	194	195	311	312	313	314	315
196	197	198	199	200	316	317	318	319	320
201	202	203	204	205	321	322	323	324	325
206	207	208	209	210	326	327	328	329	330
211	212	213	214	215	331	332	333	334	335
216	217	218	219	220	336	337	338	339	340

SAVE! SAVE! SAVE!

If AC's TECH is just what you've been looking for in an Amiga publication, don't just sit there –

SUBSCRIBE!

Also consider two more great reasons to be a subscriber – AC and AC's GUIDE! Just fill out, clip and mail this card along with your payment to pick up on the

SAVINGS!

YES! The Amazing AC publications give me **3 GREAT** reasons to save! Please begin the subscription(s) indicated below immediately!

Name_____

Address _____

City_____ State _____ ZIP_____

Charge my ☐ Visa ☐ MC #_____

Expiration Date _____ Signature _____

Please circle to indicate this is a **New Subscription** or a **Renewal**

		US $39.95 ☐
1 year of **AC's TECH**	**4 big issues of AC's TECH!** Limited Time Charter Offer!	Canada/Mexico $43.95 ☐
		Foreign Surface $47.95 ☐
1 year of **AC**	**12 big issues of Amazing Computing!** Save over 49% off the cover price!	US $24.00 ☐
		Canada/Mexico $34.00 ☐
		Foreign Surface $44.00 ☐
1-year SuperSub	**AC + AC's GUIDE – 14 issues total!** Save more than $31 off the cover prices!	US $36.00 ☐
		Canada/Mexico $54.00 ☐
		Foreign Surface $64.00 ☐
2 years of **AC**	24 big issues! Save over 59%! US only.	US $38.00 ☐
2-year SuperSub	28 big issues! Save more than $75! US only.	US $59.00 ☐

Check or money order payments must be in US funds drawn on a US bank; subject to *applicable* sales tax.

MasterCard

VISA

Please return to:

AC's TECH AMIGA

P.i.M. Publications, Inc.
P.O. Box 869
Fall River, MA 02722-0869

Please place this order form in an envelope
with your check or money order.